

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

基于Lambda架构的系统构建

Big Data

Principles and Best Practices of
Scalable Realtime Data Systems

大数据系统构建

可扩展实时数据系统构建
原理与最佳实践

[美] 南森·马茨 (Nathan Marz) 著
詹姆斯·沃伦 (James Warren)

马延辉 向磊 魏东琦 译



机械工业出版社
China Machine Press

随着社交网络、网络分析和智能型电子商务的兴起，传统的数据库系统显然已无法满足海量数据的管理需求。作为一种新的处理模式，大数据系统应运而生，它使用多台机器并行工作，能够对海量数据进行存储、处理、分析，进而帮助用户从中提取对优化流程、实现高增长率的有用信息，做更为精准有效的决策。但不可忽略的是，它也引入了大多数开发者并不熟悉的、困扰传统架构的复杂性问题。

本书将教你充分利用集群硬件优势的Lambda架构，以及专门用来捕获和分析网络规模数据的新工具，来创建这些系统。它将描述一个可扩展的、易于理解大数据系统的方法——可以由小团队构建并运行。本书共18章，除了介绍基本概念，其他章节采用“理论+示例”的方式来阐释相关概念，并使用现实世界中的工具加以论证。其中，第1章介绍了数据系统的原理，给出了Lambda架构的概述，并概述了构建任何数据系统的广义方法。第2~9章集中阐述Lambda架构的批处理层。第10章和第11章集中阐述服务层，让读者了解只批量写入的特定数据库——这些数据库比传统数据库更简单，它们具有出色的性能，并具备可操作性、稳健性等特点。第12~17章集中阐述速度层，让读者更明确地了解NoSQL数据库、流处理和管理增量计算的复杂性。第18章通过综合回顾Lambda架构的相关知识，帮助读者了解增量批处理、基本Lambda架构的变种，以及如何充分利用资源。

大数据

技术丛书

Big Data

Principles and Best Practices of
Scalable Realtime Data Systems

大数据系统构建

可扩展实时数据系统构建
原理与最佳实践

[美] 南森·马茨 (Nathan Marz) 著
詹姆斯·沃伦 (James Warren)

马延辉 向磊 魏东琦 译



机械工业出版社
China Machine Press

图书在版编目(CIP)数据

大数据系统构建:可扩展实时数据系统构建原理与最佳实践/(美)南森·马茨(Nathan Marz), (美)詹姆斯·沃伦(James Warren)著;马延辉,向磊,魏东琦译. —北京:机械工业出版社, 2016.12

(大数据技术丛书)

书名原文: Big Data: Principles and Best Practices of Scalable Realtime Data Systems

ISBN 978-7-111-55294-9

I. 大… II. ①南… ②詹… ③马… ④向… ⑤魏… III. 数据处理 IV. TP274

中国版本图书馆CIP数据核字(2016)第262539号

本书版权登记号:图字:01-2015-7585

Nathan Marz, James Warren: Big Data: Principles and Best Practices of Scalable Realtime Data Systems (ISBN 978-1617290343).

Original English language edition published by Manning Publications Co., 209 Bruce Park Avenue, Greenwich, Connecticut 06830.

Copyright © 2015 by Manning Publications Co.

Simplified Chinese-language edition copyright © 2017 by China Machine Press.

Simplified Chinese-language rights arranged with Manning Publications Co. through Waterside Productions, Inc.

No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or any information storage and retrieval system, without permission, in writing, from the publisher.

All rights reserved.

本书中文简体字版由Manning Publications Co. 通过Waterside Productions, Inc. 授权机械工业出版社在全球独家出版发行。未经出版者书面许可, 不得以任何方式抄袭、复制或节录本书中的任何部分。

大数据系统构建

可扩展实时数据系统构建原理与最佳实践

出版发行:机械工业出版社(北京市西城区百万庄大街22号 邮政编码:100037)

责任编辑:吴晋瑜

责任校对:殷虹

印刷:北京诚信伟业印刷有限公司

版次:2017年1月第1版第1次印刷

开本:186mm×240mm 1/16

印张:18.75

书号:ISBN 978-7-111-55294-9

定价:79.00元

凡购本书,如有缺页、倒页、脱页,由本社发行部调换

客服热线:(010) 88379426 88361066

投稿热线:(010) 88379604

购书热线:(010) 68326294 88379649 68995259

读者信箱:hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问:北京大成律师事务所 韩光/邹晓东

The Translator's Words 译者序

首先, 请允许我们对 Nathan Marz 致以崇高的敬意。

Nathan Marz 是分布式实时计算系统 Storm 的创始人, 在 Twitter 收购社交媒体数据分析公司 BackType 前担任 BackType 的首席工程师, 之后选择离开 Twitter, 创立自己的公司。在实时大数据处理系统中, Storm 作为 Apache 顶级开源项目已经成为大数据界不可或缺的一部分。因此, 对于能够翻译 Nathan Marz 的书籍, 我们深感荣幸。

与大多数程序员一样, Nathan Marz 也是通过游戏进入开发者的世界的, 在这一点上, 似乎我们大多数人与 Nathan Marz 相差无几。但不同的是, Nathan Marz 开创性地设计并使用 Clojure 语言编写了 Storm, 为我们揭开了大数据处理的新篇章, 而我们未曾想过海量数据是可以实时分析并处理的, 这也正是他与众不同的地方。Nathan Marz 对大数据概念的理解非常深刻, 在编程技术上基础扎实, 如同 Dean Jeffrey 和 Doug Cutting 那样, 他用自己的超凡的智慧, 带领我们步入了一个全新的数据时代。

本书借一些虚构的社交媒体示例, 来让读者深入理解以下几件事情:

- 1) 什么是大数据, 它们从哪里来?
- 2) 社交媒体有哪些数据是有价值且需要我们去分析的?
- 3) 在使用数据的过程中, 我们需要用哪些思路、架构、工具来实现自己的目的?
- 4) 对于不同的数据类型, 我们如何选择正确的架构和模型去进行分析和挖掘?

在翻译的过程中, 我们也了解到, Nathan Marz 不仅在数学与编程方面才华横溢, 对各种开发工具与架构也是信手拈来, 而且他所写的书籍也是字字珠玑, 文不加点。他所写的内容深邃却并不晦涩, 浅显易懂, 贴近实战, 原作行文流畅, 文采炳焕。本书将大数据方方面面的工具以实例的形式引入内容中, 令人读后有一种酣畅淋漓、耳目一新的感觉, 在内容方面, 从 Apache Thrift 的讲解到 Lambda 架构的实例、从 HDFS 和 MapReduce 的示范到架构和算法的实现以及针对不同类型数据模型的创建, 一一涵盖其中。可以说, 本书是

大数据技术的集大成者，是诸多大数据书籍中难得一见的实战参考书。

对于我们译者来说，之所以翻译本书，既是希望将国外实践大数据技术的重要经验引入国内，让国内的读者能够从中一窥究竟，同时也希望自己在翻译的过程中有所受益。站在巨人的肩膀上，才能让我们能够看得更远。

在本书的翻译过程中，我们得到了诸多朋友和家人的帮助、理解以及支持，在此对他们表示衷心的感谢。同时也对促成本书出版的机械工业出版社的王春华、杨福川编辑表示诚挚的谢意。

本书内容丰富，涵盖了大数据的诸多方面，如 Thrift、数据建模、HDFS、MapReduce、HBase、Lambda 等，这为本书的翻译增加了不少难度。尽管我们进行了多次校对和修改，甚至几位译者就某些专业词汇如何准确翻译进行了多次字斟句酌的讨论，但由于水平所限，恐难以将原作的内容全面还原，因此也难免出现纰漏和不足。在此，也恳请广大读者在阅读之余不吝赐教，给予批评指正。

向 磊

2016 年 10 月于北京

Preface 前言

当第一次进入大数据的世界时，我仿佛置身于软件开发的美国西部荒原。许多人放弃了关系型数据库，转而选择带有高度受限模型的 NoSQL 数据库，主要是因为其使用体验良好、熟悉度较高且这种数据库可以扩展到成千上万台机器上。NoSQL 数据库的数量巨大，堪称铺天盖地，这些数据库中很多都只有细微的差别。一个名为“Hadoop”的新项目开始崭露头角，它宣称具备基于海量数据进行数据深度分析的能力。但弄清楚如何使用这些新工具很令人困惑。

当时，我正试图处理所在公司面临的扩展性问题。系统架构非常复杂——该 Web 系统包含共享关系型数据库、队列、工作节点、主节点和从节点。数据损坏渗透至数据库，为了处理这些损坏，我们使用了应用程序中的特殊代码，但从节点的操作总是落后于其他节点。我决定探索其他大数据技术，看看是否有比我们的数据架构更好的设计。

早期的软件工程职业生涯的经历，深刻影响了我对“系统该如何架构”的观点。我的一位同事花了几个星期将来自互联网的数据收集到一个共享文件系统。他在等待收集足够的数据，以便能在其上进行分析。有一天，在做一些日常维护时，我不小心删除了他的所有数据，导致他的项目延期了好几周。

我知道自己犯了一个大错，但作为一个软件工程师新手，我并不知道这会导致什么样的后果。我会不会因为粗心被解雇呢？我发了一封电子邮件向团队诚挚地道歉——让我惊喜的是，大家对此都表示非常同情。我永远不会忘记那个时刻——一个同事来到我的办公桌旁，拍着我的背说：“恭喜你！你现在是一个专业的软件工程师了！”

他玩笑式的表述道出了软件开发中不言而喻的“真理”——我们不知道如何创造完美的软件。软件可能有 bug 而且会被部署到生产中。如果应用程序可以写入数据库中，那么 bug 也可能写入数据库中。当着手重新设计我们的数据架构时，这样的经历深深地影响了我。我知道，新架构不但必须是可扩展的、对机器故障是可容错的，并且要易于推断故障

原因——但对人为错误也可容错。

重构那套系统的经验，促使我走上了一条“在数据库和数据管理方面怀疑一切我认为正确的”道路。我想出了一个基于不可变数据和批量计算的架构，令我很惊讶的是，与仅仅基于增量计算的系统相比，新系统要简单得多。一切都变得更容易，包括操作、不断发展的系统以支持新的功能、从人为错误中恢复和性能优化等方面。该方法很通用，似乎可以用于任何数据系统。

但有些事情困扰着我。当观察其他行业时，我发现几乎没有人使用类似的技术。相反，在使用基于增量更新数据库的庞大集群架构中，令人生畏的复杂性是为人所接受的。这些架构的许多复杂性已经通过我所开发的方法完全避免或大大缓减了。

在接下来的几年中，我扩展了该方法，并使之正式成为我戏称的 **Lambda 架构**。在初创公司 BackType 工作时，我们的 5 人团队构建了一个社会化媒体分析产品，该产品支持在超过 100TB 的数据上进行多样化实时分析。我们的小团队还负责拥有数百台机器的集群的管理部署、运营和系统监控。当我们向别人展示自己的产品时，他们对这个团队只有 5 个人感到非常惊讶。他们经常会问“这么几个人做了这么多事情？怎么可能！？”我的回答很简单：“不是我们在做什么，而是我们没有做什么。”通过使用 Lambda 架构，我们避免了困扰传统架构的复杂性。通过避免这些复杂性，我们大大提高了工作效率。

大数据运动只是放大了已经存在了几十年的数据架构的复杂性。主要基于增量更新的大型数据库架构将遭受这些复杂性的折磨，从而导致错误、繁重的操作，并阻碍了生产力。尽管 SQL 和 NoSQL 数据库通常被描述成对立或相互对偶的关系，但从最基本的方面来说，它们实际上是一样的。它们都鼓励使用这种相同的架构——该架构具有不可避免的复杂性。复杂性是一个邪恶的野兽，无论你承认与否，它都会“咬”你。

为了传播 Lambda 架构以及它如何避免传统架构的复杂性等知识，我写了本书。它是我开始从事大数据工作时就希望有的。我希望你把这本书作为一个旅程——挑战你以为自己已经知道的关于数据系统的知识，并发现从事大数据工作也可以优雅、简单和有趣。

Nathan Marz

About This Book 关于本书

类似社交网络、网络分析和智能型电子商务这样的服务，通常需要在非常大规模的传统数据库上管理数据。复杂性随着规模与需求的增加而增加，而处理大数据并不是简单地将 RDBMS 扩大一倍或推出一些时髦的新技术。幸运的是，可扩展性和简单性并不是相互排斥的——你只需要采取不同的方法。大数据系统使用多台机器并行工作来存储和处理数据，它引入了大多数开发者并不熟悉的根本性的挑战。

本书将教你充分利用集群硬件优势的架构，以及专门用来捕获和分析网络规模数据的新工具，来创建这些系统。它将描述一个可扩展的、易于理解大数据系统的方法，可以由小团队构建并运行。本书利用一个实际示例，基于大数据系统的理论在实践中实现它们来指导读者。

本书不要求读者以前接触过大规模数据分析或 NoSQL 工具。熟悉传统数据库是有帮助的，但不是必需的。本书旨在教你如何思考数据系统，以及如何化繁为简。我们将从基本原理开始，从那些被认定为架构的各个组件所必需的属性开始。

路线图

本书包括 18 章，各章的主要内容如下。

第 1 章介绍了数据系统的原理，并给出了 Lambda 架构的概述：构建任何数据系统的广义方法。第 2~17 章以理论和示例交替讲解的方式深入介绍 Lambda 架构的所有内容。理论章节阐述了若干概念，这些概念对现有工具都是适用的；同时，示例章节使用现实世界中的工具来论证这些概念。不要被名字迷惑，虽然所有章节都是样例驱动。

第 2~9 章集中阐述 Lambda 架构的批处理层。在这里，你将了解如何为主数据集建模、如何使用批处理来创建数据的任意视图，以及如何进行增量和批处理之间的权衡。

第 10 章和第 11 章集中阐述服务层，它支持低延迟访问由批处理层中产生的视图。在这里，你将了解只批量写入的特定数据库。你将发现，这些数据库比传统数据库更简单，它们具有出色的性能，并具备可操作性、稳健性等特性。

第 12~17 章集中阐述速度层，该层弥补了批处理层的高延迟，为所有查询提供最新结果。在这里，你将了解 NoSQL 数据库、流处理和管理增量计算的复杂性。

第 18 章再次复习 Lambda 架构的相关知识，并进行查漏补缺。你将了解增量批处理、基本 Lambda 架构的变种，以及如何充分利用资源。

代码下载和约定

本书的源代码可以在 <https://github.com/Big-Data-Manning> 找到。我们提供了运行示例 SuperWebAnalytics.com 的源代码。

大量源代码以代码清单的形式给出。这些代码清单提供了完整的代码段。一些代码带有注释，以着重强调或解释某部分代码。在正文的其他地方，代码片段会在必要时使用。Courier 字体用来表示 Java 代码。在代码中，我们用粗体字来帮助你识别文本中的关键部分。

作者在线

购买本书的读者将可免费使用由 Manning 出版社运营的私人网络论坛。你可以在论坛上评论本书、提出技术问题，并从作者和其他用户处得到帮助。要访问和订阅该论坛，请在 Web 浏览器输入“www.manning.com/BigData”。在论坛上注册后，你可在作者在线 (AO) 页面查看如下信息：注册后如何登录论坛、哪些帮助可用以及论坛上的行为规则。

Manning 承诺为读者提供一个场所，以供个体读者之间、读者和作者之间进行有意义的对话。这并不是作者承诺进行任何特定数量的分享，作者对 AO 论坛的贡献是自愿的（无报酬的）。我们建议你尝试问作者一些有挑战性的问题，以免他们觉得了然无趣！

只要本书已出版，AO 论坛和以前讨论的归档文件即可以从发行商的网站访问。

关于封面插图

本书的封面插图是“Le Raccommodeur de Fiance”，意思是泥瓦匠。泥瓦匠擅长修补破

损或有缺口的盆、盘、杯和碗，他走过法国的城镇和村庄，做着自己的生意。

该图摘自 19 世纪法国出版的 Sylvain Maréchal 四卷本的地域服饰风俗纲要。其中每幅图都是精心绘制并手工着色的。Maréchal 丰富多彩的收藏向人们生动地展示了 200 年前城市 and 地区在文化上的差异——相互隔离，人们操着不同的方言和语言。在街头或是在农村，只是通过他们的着装就能够很容易地辨认他们生活的地方和他们的职业或生活中的地位。

自那时以来，着装规范已经发生改变，地区的多样性那时候是如此丰富，但这种多样性现在却消失了。现在很难分辨不同大陆的居民，更别说不同城市或地区的居民了。也许我们是用更多样化的个人生活取代了文化的多样性——现在肯定是更多样化和快节奏的科技化生活。

为了与其他计算书籍区别开来，Manning 希望通过这样的封面插图来庆祝计算机业务的创造性和主动性，所以又将 Maréchal 的图片带回大众视野。

致谢 Acknowledgements

如果没有周围很多人的帮助，这本书是无法完成的。我必须首先感谢我的父母，他们一直给我灌输热爱学习和探索世界的思想，并一直鼓励我在职业生涯中孜孜不倦地追求。

我的哥哥 Iorav 也一直在学术兴趣上给予我鼓励。我还记得，在我读小学时，他就教我学习代数。我第一次接触编程也是他介绍的——他教我 Visual Basic，因为他在高中学过该课程。这些课程引发了我对编程的热情，引导我走上了职业道路。

非常感谢 Michael Montano 和 Christopher Golda——BackType 的创始人。从他们把我作为他们的第一个员工开始，我就拥有可以自己做决定的极度自由。这种自由对我探索和最大限度地利用 Lambda 架构来说至关重要。他们从未质疑过开源的价值，并允许我自由地开源我们的技术。深入地参与开源已经成为我的特权之一。

特别感谢我在斯坦福学习时遇到的很多教授。Tim Roughgarden 是我见过的最好的老师——他从根本上提高了我严格分析、解构和解决困难问题的能力。尽可能多地上他的课是我一生中做出的最好决定之一。也感谢 Monica Lam 给我灌输了对 Datalog 优雅性的欣赏。许多年后我将 Datalog 与 MapReduce 结合，生成了平生第一个意义重大的开源项目——Cascalog。

Chris Wensel 是第一个给我展示大规模的数据处理也有可能优雅和高效的人。他的 Cascading 库改变了我看待大数据处理的方式。

如果没有大数据领域的先驱者，我的工作是不可能实现的。特别感谢最早的 MapReduce 论文的作者 Jeffrey Dean 和 Sanjay Ghemawat。感谢最初的 Dynamo 论文的作者 Giuseppe DeCandia、Deniz Hastorun、Madan Jampani、Gunavardhan Kakulapati、Avinash Lakshman、Alex Pilchin、Swaminathan Sivasubramanian、Peter Vossahl 和 Werner Vogels。感谢 Apache Hadoop 项目的创始人 Michael Cafarella 和 Doug Cutting。

在我的编程生涯中，Rich Hickey 一直是给我最多灵感的人之一。Clojure 是我至今用过的最好的语言，通过学习它，我成了一名更好的程序员。我很欣赏它的实用性和专注于简单性的特性。Rich 在编程的状态和复杂性理念方面已经深深地影响了我。

开始写这本书时，我几乎称不上是作者。Renae Gregoire，Manning 公司负责本书的策划编辑之一，特别感谢她帮助我提高写作技巧。她让我认识到使用例子来引入通用概念的重要性。在如何有效地组织技术写作方面，她给了我很多灵感。她教给我的技巧不仅适用于写作技术书籍，还适用于写博客、演讲和平时的沟通。因为掌握了一项重要的生活技能，所以我对她永远心存感激。

如果没有我的合著者 James Warren 的努力，本书将不会有现在的质量。为了让读者能够理解理论概念，并找到展示这些理论的更好方式，他做了大量的工作。本书之所以能够如此明晰，大部分是源于他强大的沟通技巧。

与我的出版商 Manning 合作是一种乐趣。他们的员工对我很耐心，并且理解寻找合适的方式写出这样一个大的话题是需要时间的。在整个过程中，他们都很支持我并帮助我，他们总是提供给我成功所需的资源。感谢 Marjan Bace 和 Michael Stephens 的支持，还有所有其他员工的帮助和全程指导。

我尝试尽可能多地学习其他作家的写作风格。Bradford Cross、Clayton Christensen、Paul Graham、Carl Sagan 和 Derek Sivers 对我的影响很大。

最后，十分感谢对本书提出意见、评论和反馈的数百余名读者。这些反馈促成了本书的多次修改、重写和重组架构，直到我们找到有效地展示材料的方法。在此特别感谢 Aaron Colcord、Aaron Crow、Alex Holmes、Arun Jacob、Asif Jan、Ayon Sinha、Bill Graham、Charles Brophy、David Beckwith、Derrick Burns、Douglas Duncan、Hugo Garza、Jason Courcoux、Jonathan Esterhazy、Karl Kuntz、Kevin Martin、Leo Polovets、Mark Fisher、Massimo Ilario、Michael Fogus、Michael G. Noll、Patrick Dennis、Pedro Ferrera Bertran、Philipp Janert、Rodrigo Abreu、Rudy Bonefas、Sam Ritchie、Siva Kalagarla、Soren Macbeth、Timothy Chklovski、Walid Farid 和 Zhenhua Guo。

Nathan Marz

在回想为本书做出贡献的人时，我很震惊这么多人帮助过我。虽然不能一一列举，但这并不能减轻我的谢意。尽管如此，我还是希望向一些人明确表达我的感激之情：

□ 我的妻子 Wen-Ying Feng——感谢你的爱、鼓励和支持，不仅是这本书，还有我们

共同完成的一切。

- 我的父母 James 和 Gretta Warren——感谢你们带给我的无穷信仰，还有为我提供每个机会所做出的牺牲。
- 我的姐姐 Julia Warren-Ulanch——感谢你树立了一个光辉的榜样，使我可以跟随你的脚步。
- 我的两位导师 Ellen Toby 和 Sue Geller 教授——感谢你们悉心回答我的每个问题，并教导我学习的乐趣不仅源于获得知识，更在于分享知识。
- Chuck Lam——感谢你很多年前对我说：“嘿，你听说过一个叫 Hadoop 的东西吗？”
- 我的朋友和 RockYou!、Storm8、Bina 的同事——感谢我们一起共享的经历和把理论运用到实践的机会。
- Marjan Bace、Michael Stephens、Jennifer Stout、Renae Gregoire 和整个 Manning 的编辑出版人员——感谢你们在本书出版过程中的指导和耐心。
- 本书的评论者和早期读者——感谢你们的评论和批评，推动我们更加清晰地阐述理论；感谢你们让这本书变得更好。

最后，我要对 Nathan 表达最真挚的谢意，感谢你邀请我一起完成本书。在加入这个项目之前，我已经非常仰慕你的工作，共事时因为你的想法和理念，使我更加尊重你。这是一项莫大的荣誉和特权。

James Warren

Contents 目 录

译者序	
前言	
关于本书	
致谢	

第1章 大数据的新范式 1

1.1 本书是如何组织的	2
1.2 扩展传统数据库	3
1.2.1 用队列扩展	3
1.2.2 通过数据库分片进行扩展	4
1.2.3 开始处理容错问题	4
1.2.4 损坏问题	5
1.2.5 到底是哪里出错了	5
1.2.6 大数据技术是如何起到帮助作用的	5
1.3 NoSQL 不是万能的	6
1.4 基本原理	6
1.5 大数据系统应有的属性	7
1.5.1 鲁棒性和容错性	7
1.5.2 低延迟读取和更新	8
1.5.3 可扩展性	8
1.5.4 通用性	8
1.5.5 延展性	8

1.5.6 即席查询	9
1.5.7 最少维护	9
1.5.8 可调试性	9
1.6 全增量架构的问题	10
1.6.1 操作复杂性	10
1.6.2 实现最终一致性的极端复杂性	11
1.6.3 缺乏容忍人为错误	12
1.6.4 全增量架构解决方案与 Lambda 架构解决方案	13
1.7 Lambda 架构	14
1.7.1 批处理层	15
1.7.2 服务层	16
1.7.3 批处理层和服务层满足几乎所有属性	16
1.7.4 速度层	17
1.8 技术上的最新趋势	19
1.8.1 CPU 并不是越来越快	20
1.8.2 弹性云	20
1.8.3 大数据充满活力的开源生态系统	20
1.9 示例应用: SuperWebAnalytics.com	21
1.10 总结	22

第一部分 批处理层

第2章 大数据的数据模型 24

2.1 数据的属性 25

2.1.1 数据是原始的 28

2.1.2 数据是不可变的 30

2.1.3 数据是永远真实的 33

2.2 基于事实的数据表示模型 34

2.2.1 事实的示例及属性 34

2.2.2 基于事实的模型的优势 36

2.3 图模式 39

2.3.1 图模式的元素 39

2.3.2 可实施模式的必要性 40

2.4 SuperWebAnalytics.com 的完整 数据模型 41

2.5 总结 42

第3章 大数据的数据模型：示例 44

3.1 为什么使用序列化框架 44

3.2 Apache Thrift 45

3.2.1 节点 46

3.2.2 边 46

3.2.3 属性 47

3.2.4 把一切组合成数据对象 47

3.2.5 模式演变 48

3.3 序列化框架的局限性 49

3.4 总结 50

第4章 批处理层的数据存储 51

4.1 主数据集的存储需求 52

4.2 为批处理层选择存储方案 53

4.2.1 使用键 / 值存储主数据集 53

4.2.2 分布式文件系统 54

4.3 分布式文件系统是如何工作的 54

4.4 使用分布式文件系统存储主数据集 56

4.5 垂直分区 58

4.6 分布式文件系统的底层性质 58

4.7 在分布式文件系统上存储 SuperWebAnalytics.com 的 主数据集 60

4.8 总结 61

第5章 批处理层的数据存储：

示例 62

5.1 使用 HDFS 62

5.1.1 小文件问题 64

5.1.2 转向更高层次的抽象 64

5.2 使用 Pail 在批处理层存储数据 65

5.2.1 Pail 基本操作 66

5.2.2 序列化对象到 Pail 中 67

5.2.3 使用 Pail 进行批处理操作 69

5.2.4 使用 Pail 进行垂直分区 69

5.2.5 Pail 文件格式与压缩 71

5.2.6 Pail 优点的总结 71

5.3 存储 SuperWebAnalytics.com 的 主数据集 72

5.3.1 Thrift 对象的结构化 Pail 73

5.3.2 SuperWebAnalytics.com 的基础 Pail 74

5.3.3 用于垂直分区数据集的分片 Pail 75

5.4 总结 78

第6章 批处理层..... 79

6.1 启发性示例..... 80

6.1.1 给定时间范围内的页面浏览量..... 80

6.1.2 性别推理..... 80

6.1.3 影响力分数..... 81

6.2 批处理层上的计算..... 82

6.3 重新计算算法与增量算法..... 84

6.3.1 性能..... 85

6.3.2 容忍人为错误..... 86

6.3.3 算法的通用性..... 86

6.3.4 选择算法的风格..... 87

6.4 批处理层中的可扩展性..... 87

6.5 MapReduce: 一种大数据计算的 范式..... 88

6.5.1 可扩展性..... 89

6.5.2 容错性..... 91

6.5.3 MapReduce 的通用性..... 92

6.6 MapReduce 的底层特性..... 94

6.6.1 多步计算很怪异..... 94

6.6.2 手动实现连接非常复杂..... 94

6.6.3 逻辑和物理执行紧密耦合..... 96

6.7 管道图——一种关于批处理计算的 高级思维方式..... 97

6.7.1 管道图的概念..... 97

6.7.2 通过 MapReduce 执行管道图..... 101

6.7.3 合并聚合器..... 101

6.7.4 管道图示例..... 102

6.8 总结..... 103

第7章 批处理层: 示例..... 104

7.1 一个例证..... 105

7.2 数据处理工具的常见陷阱..... 106

7.2.1 自定义语言..... 107

7.2.2 不良的可组合抽象..... 107

7.3 JCascalog 介绍..... 108

7.3.1 JCascalog 的数据模型..... 109

7.3.2 JCascalog 查询的结构..... 110

7.3.3 查询多个数据集..... 111

7.3.4 分组和聚合器..... 113

7.3.5 对一个查询示例进行单步调试..... 114

7.3.6 自定义谓词操作..... 117

7.4 组合..... 121

7.4.1 合并子查询..... 122

7.4.2 动态创建子查询..... 123

7.4.3 谓词宏..... 125

7.4.4 动态创建谓词宏..... 128

7.5 总结..... 130

第8章 批处理层示例: 架构和算法..... 131

8.1 SuperWebAnalytics.com 批处理层的 设计..... 132

8.1.1 所支持的查询..... 132

8.1.2 批处理视图..... 132

8.2 工作流概述..... 135

8.3 获取新数据..... 137

8.4 URL 规范化..... 137

8.5 用户标识符规范化..... 138

8.6 页面浏览去重..... 142

8.7 计算批处理视图..... 142

8.7.1 给定时间范围内的页面 浏览量..... 143

8.7.2 给定时间范围内的独立 访客..... 143

8.7.3 跳出率分析..... 144

8.8 总结	145
--------	-----

第9章 批处理层示例：实现

9.1 出发点	147
9.2 准备工作流	148
9.3 获取新数据	149
9.4 URL 规范化	152
9.5 用户标识符规范化	153
9.6 页面浏览去重	159
9.7 计算批处理视图	159
9.7.1 给定时间范围内的页面浏览量	159
9.7.2 给定时间范围内的独立访客	161
9.7.3 跳出率分析	163
9.8 总结	165

第二部分 服务层

第10章 服务层概述

10.1 服务层的性能指标	169
10.2 规范化 / 非规范化问题的服务层解决方案	172
10.3 服务层数据库的需求	173
10.4 设计 SuperWebAnalytics.com 的服务层	174
10.4.1 给定时间范围内的页面浏览量	175
10.4.2 给定时间范围内的独立访客	175
10.4.3 跳出率分析	176
10.5 对比全增量的解决方案	177
10.5.1 给定时间范围内的独立访客的全增量方案	177

10.5.2 与 Lambda 架构解决方案的比较	182
---------------------------	-----

10.6 总结	183
---------	-----

第11章 服务层：示例

11.1 ElephantDB 的基本概念	184
11.1.1 ElephantDB 中的视图创建	185
11.1.2 ElephantDB 中的视图服务	185
11.1.3 使用 ElephantDB	186
11.2 创建 SuperWebAnalytics.com 的服务层	188
11.2.1 给定时间范围内的页面浏览量	188
11.2.2 给定时间范围内的独立访客数量	191
11.2.3 跳出率分析	191
11.3 总结	192

第三部分 速度层

第12章 实时视图

12.1 计算实时视图	195
12.2 存储实时视图	197
12.2.1 最终一致性	198
12.2.2 速度层中存储的状态总量	198
12.3 增量计算的挑战	199
12.3.1 CAP 原理的有效性	199
12.3.2 CAP 原理和增量算法之间复杂的相互作用	201
12.4 异步更新与同步更新	202
12.5 过期实时视图	203

12.6 总结	205	16.1.3 微批量流处理的拓扑结构	242
第 13 章 实时视图：示例	206	16.2 微批量流处理的核心概念	244
13.1 Cassandra 的数据模型	206	16.3 微批量流处理的扩展管道图	245
13.2 使用 Cassandra	208	16.4 完成 SuperWebAnalytics.com 的 速度层	246
13.3 总结	210	16.4.1 给定时间范围内的页面 浏览量	246
第 14 章 队列和流处理	211	16.4.2 跳出率分析	247
14.1 队列	211	16.5 另一个跳出率分析示例	251
14.1.1 单消费者队列	212	16.6 总结	252
14.1.2 多消费者队列	214	第 17 章 微批量流处理：示例	253
14.2 流处理	214	17.1 使用 Trident	253
14.2.1 队列和工作节点	215	17.2 完成 SuperWebAnalytics.com 的 速度层	257
14.2.2 队列和工作节点的缺陷	216	17.2.1 给定时间范围内的页面 浏览量	257
14.3 更高层次的一次一个的流处理	217	17.2.2 跳出率分析	259
14.3.1 Storm 模型	217	17.3 完全容错、基于内存及微批量 处理	265
14.3.2 保证消息处理	221	17.4 总结	266
14.4 SuperWebAnalytics.com 速度层	223	第 18 章 深入 Lambda 架构	268
14.5 总结	226	18.1 定义数据系统	268
第 15 章 队列和流处理：示例	227	18.2 批处理层和服务层	270
15.1 使用 Apache Storm 定义拓扑结构	227	18.2.1 增量的批处理	270
15.2 Apache Storm 集群及其部署	230	18.2.2 测量和优化批处理层的 资源使用	276
15.3 保证消息处理	232	18.3 速度层	280
15.4 实现 SuperWebAnalytics.com 给定 时间范围内的独立访客的速度层	233	18.4 查询层	281
15.5 总结	237	18.5 总结	282
第 16 章 微批量流处理	239		
16.1 实现有且仅有一次语义	240		
16.1.1 强有序处理	240		
16.1.2 微批量流处理	241		

大数据的新范式

本章内容

- ❑ 扩展传统数据库时遇到的典型问题
- ❑ 为什么 NoSQL 不是万能的
- ❑ 从基本原理思考大数据系统
- ❑ 大数据工具的情形
- ❑ SuperWebAnalytics.com 的介绍

在过去的十年里，人们创造的数据量一路飙升——每秒产生超过 30 000GB 的数据，并且数据创造的速度仍在加快。

我们处理的数据是多样化的。用户创建的内容有博客文章、微博、社交网络的互动和照片等。服务器不断地记录用户正在做什么的消息。科学家们创造了精确“测量”世界的标尺——周围世界的详细测量结果。互联网，这使得数据的最终来源是浩瀚无垠的。

数据的惊人增长已经深刻地影响了商业。传统的数据库系统，比如关系型数据库，已经被发展到极限。在越来越多的情况下，这些系统已经承受不住“大数据”的压力了。传统系统以及与它们相关的数据管理技术未能成功扩展到大数据的范畴。

为了应对大数据的挑战，新一代的技术出现了。其中许多新技术被分类到 NoSQL 术语下。在某些方面，这些新技术比传统数据库更复杂，而在另外一些方面，它们更简单一些。这些系统可以扩展到非常大的数据集，但要想有效地使用这些技术，需要一套全新的

技巧——然而并没有放之四海而皆准的解决方案。

这些大数据系统中有许多是由 Google 所开创的，包括分布式文件系统、MapReduce 计算框架和分布式锁服务。该领域中的另一个先驱是 Amazon，它创造了一个革新性的分布式键/值存储系统，称为“Dynamo”。这些年，开源社区开发出了如下的 Hadoop、HBase、MongoDB、Cassandra、RabbitMQ 和无数其他项目。

本书是关于复杂性和可扩展性的。为了应对大数据的挑战，我们将从头开始重新考虑数据系统。你会发现在例如关系型数据库管理系统（Relational Database Management Systems, RDBMS）这样的传统系统中，人们管理数据的一些最基本方法对于大数据系统来说过于复杂。为简单起见，替代方法就是你即将探索的大数据新范式。我们把这种方法称为“Lambda 架构”。

在第 1 章中，你将探索“大数据问题”以及为什么需要大数据的新范式。你会看到一些传统技术扩展的风险，并发现用传统方式构建数据系统的缺陷。以数据系统基本原理为出发点，我们将制订一种不同的方式来构建数据系统，以避免传统技术的复杂性。你会看到技术的最近趋势是如何促进新系统的使用的，最后你还会看到一个大数据系统的例子，我们将在整本书中贯穿构建这个大数据系统，以此演示关键概念。

1.1 本书是如何组织的

你可以认为本书主要是一本理论性的书籍，专注于如何构建适用于任何大数据问题的解决方案。无论目前情形下的工具怎样，你将学习的原则都是有效的，你可以据此来严格选择适合你的应用程序的工具。

本书不是对数据库、计算和其他相关技术的调研。尽管在本书中，你将学习如何使用诸如 Hadoop、Cassandra、Storm 和 Thrift 等工具，但是本书的目的不是学会使用这些工具。相反，这些工具是学习构建具备鲁棒性和可扩展性的数据系统的一种手段。在这些工具之间进行相关的比较和对比，对你来说是不公平的，这样就会偏离学习底层原理的道路。换句话说，你将学习如何钓鱼，而不仅仅是如何使用特定的鱼竿。

通过这种方式，我们将本书划分为理论章节和例证章节。你可以只阅读理论章节，以全面理解构建大数据系统的方法——但我们认为，将理论映射到例证章节中的特定工具这一过程，将会使你对这些资料有更丰富、更细致的理解。

但是不要被本书的名字所欺骗——理论章节也有很多示例。本书中的首要案例是 SuperWebAnalytics.com，这一案例在理论和例证章节都有用到。在理论章节，你可以看到 SuperWebAnalytics.com 的算法、索引设计和架构。例证章节利用具体的工具将这些设计映

射为功能代码。

1.2 扩展传统数据库

我们将从许多开发人员的“起点”开始大数据的探索，直击传统数据库技术的局限性。

假设上司的要求是构建一个简单的网络分析应用程序。这个应用程序能追踪客户期望追踪的任何统一资源定位符（Uniform Resource Locator, URL）的页面浏览量。每接收到一次页面浏览，客户的网页就用其 URL ping 应用程序的 Web 服务器。此外，应用程序在什么时候都能根据页面浏览量给出前 100 排名的 URL。

首先启动一个如图 1-1 所示的页面浏览量的传统关系型模式。其后端包括一个该模式的表的 RDBMS 和一个 Web 服务器。每当有人加载被应用程序追踪的网页时，这一网页带着页面浏览 ping Web 服务器，同时 Web 服务器会在数据库中增加相应的行。

列 名	类 型
id	整型
user_id	整型
url	变长字符型（255）
pageviews	长整型

图 1-1 简单分析应用程序的关系型模式

让我们看看在完善应用程序时出现了什么问题。——如你所见，这里会遇到可扩展性和复杂性问题。

1.2.1 用队列扩展

网络分析产品获得了巨大的成功，应用程序的流量正像野火一样增长。例如，公司举办了一个盛大的派对，但庆祝时，你开始从监控系统收到大量的电子邮件。这些邮件都在说同样的事情——“插入数据库时发生超时错误。”

你查看了日志，问题很明显——数据库跟不上负载，导致增加页面浏览量的写请求超时。

你需要尽快做些事情来解决这个问题。你会意识到每次只执行一次增量操作到数据库是很浪费的。如果你在单个请求中批处理多个增量操作，这样就可以更有效。所以你重构后端，使这一切成为可能。

所谓重构后端，不是让 Web 服务器直接访问数据库，而是在 Web 服务器和数据库之间插入一个队列。当你收到一个新的页面浏览后，该事件被添加到队列中；然后创建一个一次从队列中读取 100 个事件的工作进程，并在单个数据库更新操作中批量插入它们，如图 1-2 所示。



图 1-2 使用队列和工作进程实现批量更新

这个方案执行得很好，它解决了超时问题。它甚至还有额外的好处——如果数据库再次超载，只会使队列变得更大，而不会导致 Web 服务器超时和潜在的数据丢失。

1.2.2 通过数据库分片进行扩展

不幸的是，添加队列并做批量更新只是可扩展性问题的一个“创可贴”。随着应用程序日渐受欢迎，数据库会再次超载。现有的工作进程跟不上写操作的速度，所以你尝试添加更多的工作进程来并行化更新。不幸的是，这并未起到多大的作用——显然，数据库是瓶颈。

使用 Google 搜索如何扩展写操作频繁的关系型数据库，你会发现最好的方法是使用多个数据库服务器，并在所有服务器上分散该表，使每个服务器拥有该表所包含数据的一个子集。这种方式被称为水平分区或分片。这种技术通过多个机器分散写操作的负载。

这里使用的分片技术是通过分片的数量对键的散列值进行取模，以此为每个键选择分片。使用散列函数将键映射到分片，可以使键均匀分布到所有分片。接下来写一个脚本来映射单个数据库实例中的所有行，并把数据分割成四个分片。这个脚本需要一段时间来运行，所以你要关掉增加页面浏览的工作进程，以便让该脚本完成运行，否则在转换时会丢失页面浏览的增量。

最后，所有应用程序代码需要“知道”如何为每个键找到分片。这就需要将用于从配置文件中读取分片数量的数据库处理代码封装成一个库，并且重新部署所有应用程序代码。你必须修改前 100 个 URL 的查询，从每个分片中获取前 100 个 URL 并将它们合并，用来获得全局的前 100 个 URL。

随着应用程序变得越来越流行，用户不得不将数据库重新切分成更多的分片，才能跟上写操作的负载。每次重新分片会让你觉得越来越痛苦——因为有很多工作进程需要协调，而且不能只运行一个脚本进行重新分片，那样速度会很慢。你必须并行地重新分片，并且同时管理大量的活跃工作进程的脚本。如果开发者忘记更新应用程序代码与新的分片数量，则会导致大量的增量操作写入错误的分片，因此必须编写一次性脚本来手动检查数据和移动任何错位的数据。

1.2.3 开始处理容错问题

最终，系统中有如此多的分片，以至于其中一台数据库机器的硬盘频出故障。当计算机宕机时，这部分数据是不可用的。解决这个问题的方法如下：

- 更新队列或工作进程系统，将不可用分片的增量操作放入一个单独的“等待”队列，

且试图每 5min 刷新一次“等待”队列。

- 使用数据库的复制功能为每个分片添加一个从分片，所以这样就会有一个备份以防主分片出现故障。虽然客户不在从分片中执行写操作，但至少还可以在应用程序中查看状态。

开发者会有这样的想法：“早期我花费时间为客户构建新功能，现在看来我只有花费所有时间来处理读写数据的问题了。”

1.2.4 损坏问题

当运行队列或工作进程代码时，开发者在生产环境中不小心为每个 URL 部署了一个错误的网页浏览量的增量（为 2，而不是 1），直到 24h 后才意识到这个错误，但已经造成了损坏。因为无法知道哪些数据被损坏，所以每周的备份起不到帮助作用。虽然这种方式试图使系统具备可扩展性和对机器故障的可容忍性，但无法使系统具备对人为错误的应对方式。无论你怎么努力试图阻止错误的产生，它都将不可避免地在生产环境中出现。

1.2.5 到底是哪里出错了

随着简单网络分析应用程序的发展，系统不断变得越来越复杂：队列、分片、副本、重新切分的脚本等。在数据上开发应用程序，不仅需要知道数据库模式，还需要知道更多的东西。代码需要知道如何与正确的分片通信，如果出错，从错误的分片进行读或写操作将是不可避免的。

还有一个问题是，数据库对它的分布式特性不是自我感知的，因此它不能帮客户处理分片、副本和分布式查询。这种复杂性在操作数据库和开发应用程序代码的部分中都会存在。

但最严重的问题是，系统并不是针对人为错误设计的。恰恰相反，实际上，随着系统变得越来越复杂，它越来越有可能产生错误。软件中的错误是不可避免的，如果不是在研发时犯错，则可能是写了任意破坏数据的脚本。备份无疑是足够的，开发者必须仔细考虑系统的设计方法，以降低人为错误可能导致的损害。容忍人为错误不是随意的，尤其在大数据给构建应用程序添加那么多复杂性时它更加必要。

1.2.6 大数据技术是如何起到帮助作用的

我们所要学习的大数据技术，将以令人瞩目的方式解决这些可扩展性和复杂性問題。首先，为大数据所使用的数据库和计算系统是可以感知到自己的分布式特性的，所以可以

帮助客户处理诸如分片和备份这些事情。用户不会遇到不小心查询了错误的分片的场景，因为这种逻辑是内化在数据库中的。扩展时，用户只需添加节点即可，系统将会自动重新调整到新的节点。

我们将了解的另一个核心技术是“使数据不可变”，即不是存储页面浏览量作为核心数据集，而是要存储原始网页浏览信息。因为随着新页面浏览的传入，存储页面浏览量需要用户不断地改变数据集。而原始网页浏览信息是从来不会改变的。所以当出现错误时，你可能会写入损坏的数据，但至少不会破坏良好的数据。基于数据变化方面，这是一个比传统系统更为强大的容忍人为错误的保证。对于传统的数据库，用户应谨慎使用不可变数据，因为这样的数据集将以很快的速度增长。但由于大数据技术可以扩展到如此多的数据，开发者就可以用不同的方式设计系统。

1.3 NoSQL 不是万能的

过去的十年，可扩展数据系统已经取得了大量的创新，其中包括如 Hadoop 这样的大规模计算系统，如 Cassandra 和 Riak 这样的数据库。这些系统可以处理大量的数据，但是需要认真的权衡。

比如 Hadoop 可以在非常大量的数据上并行化大规模批量计算，但计算具有较高的延迟。对于任何需要低延迟结果的计算，Hadoop 是不适用的。

又如 Cassandra 这样的 NoSQL 数据库，通过提供一个比 SQL 中使用得更有限的数据模型，来实现可扩展性。压缩应用程序到这些有限的数据模型中是非常复杂的，且因为数据库是可变的，所以它们不能容忍人为错误。

这些工具单独使用时并不是万能的。但如果智能地结合使用，那么就可以生成能处理容忍人为错误和最低复杂性的任意数据问题的可扩展系统。这就是本书中将介绍的 Lambda 架构。

1.4 基本原理

为了找到正确构建数据系统的方法，你必须先了解基本原理。那么，从最基本的层面上来说，数据系统是做什么的呢？

下面以一个直观的定义切入正题——数据系统基于过去到现在的信息来回答问题。例如，社交网络资料回答诸如“这个人的名字是什么？”“这个人有多少朋友？”这样的问题；

银行账户网页回答诸如“我的当前余额是多少？”“最近我的账户发生了什么交易？”这样的问题。

数据系统不只是记录和重现信息。它们把零碎的信息结合起来生成答案。例如，银行账户余额是结合该账户上的所有交易信息得到的。

另一个重要的观察是：并非所有信息都是平等的，一些信息来自于其他信息。例如，银行账户余额源自历史交易，朋友数源自朋友列表，朋友列表源自用户资料中添加和删除朋友的总次数。

当你一直追踪信息的来源时，最终得到的是并非派生自任何事物的信息。这是最原始的信息，也就是说，你掌握的信息是真实的，只是因为它是存在的。这样的信息就被称为数据。

你也许对数据这个词有着不同的理解。通常数据与信息这个词是可以互换使用的。但在本书的剩余部分，在使用数据这个词时，所指的是一切派生得到的特殊信息。

如果一个数据系统通过查看过去的数据来回答问题，那么最通用的数据系统通过查看整个数据集来回答问题，所以数据系统最通用的定义如下：

$$\text{query} = \text{function}(\text{all data})$$

换言之，任何所能想象的数据处理都可以表示为一个函数，该函数以接收到的所有数据作为输入。请记住这个等式，因为它是你将学到的所有知识的关键。后文将反复提及这个等式。

Lambda 架构提供了一种通用的方法来实现任意数据集上的任意函数，并且让这个函数低延迟地返回结果。这并不意味着每次实现一个数据系统时，你永远使用完全相同的技术。你使用的具体技术可能基于自身需求的改变而改变。但是 Lambda 架构定义了一个一致性的方法来选择这些技术，并将它们连接在一起以满足你的需求。

下面讨论数据系统必须呈现出的属性。

1.5 大数据系统应有的属性

你应该使大数据系统努力具备尽可能多的关于复杂性和可扩展性的属性。大数据系统不仅要运行良好、有效地使用资源，还必须易于推理。下面逐一介绍这些属性。

1.5.1 鲁棒性和容错性

面对分布式系统的挑战，构建“做正确的事”的系统并非易事。尽管会遇到机器随

机出现故障、分布式数据库中一致性的复杂语义、重复的数据、并发性等问题，但系统仍需能够正确运行。这些挑战使得“推断系统在做什么”变得很难。使大数据系统具备鲁棒性的一部分工作是为了避免这些复杂性（挑战），以便你能很容易推断系统（即探索系统）。

正如之前所讨论的，系统必须是可以容忍人为错误的。这是系统中经常被忽视的属性，开发者应予以重视。在生产系统中，偶尔有人出差错是不可避免的，比如部署错误代码损坏了数据库中的值。如果你将不变性和重新计算性构建到大数据的核心系统中，那么该系统通过提供一个清晰、简单的恢复机制，就能很容易地适应人为错误。这些内容将在第2~7章中详细描述。

1.5.2 低延迟读取和更新

一方面，绝大多数的应用程序需要对读取操作有很低的延迟——这一时间通常是几毫秒到几百毫秒；另一方面，更新延迟的需求在不同应用程序之间有着很大的区别。一些应用程序需要更新操作立即传播，但其他应用程序延迟几个小时也是可以的。无论如何，大数据系统中如果需要低延迟更新，你就必须能够实现它。更重要的是，你需要在不影响系统鲁棒性的前提下，能够实现低延迟读取和更新。本书将从第12章开始介绍如何在速度层实现低延迟更新。

1.5.3 可扩展性

面对数据或负载的增加，可扩展性能够通过将资源添加到系统中来保持性能。Lambda架构在系统堆栈的所有层是水平可扩展的——扩展是通过添加更多的机器来实现的。

1.5.4 通用性

一个通用系统可以支持大多数应用程序。事实上，如果本书所述内容没有推及大多数的应用程序，那么它不会是非常有用的。因为Lambda架构是基于所有数据功能的，可以推广到所有应用程序，无论是财务管理系统、社交媒体分析、科学应用、社交网络，还是其他应用。

1.5.5 延展性

每次添加相关功能或对系统的工作方式做出一些改变时，你不需要重复劳动。具备可延展性的系统允许以最小的开发成本添加功能。

通常，一个新特性或一个现有功能的改变需要将旧数据迁移成新格式。系统具备可扩展性的要素之一就是它容易实现大规模迁移。能够轻松、快捷地完成大规模迁移，是你将学习的方法的核心。

1.5.6 即席查询

几乎每个大型数据集中都有意料之外的值，因此，能够对数据进行即席（ad hoc）查询是非常重要的。此外，能够任意地挖掘数据集为业务优化和新的应用程序也提供了可能。最终，除非你可以问这些数据任意问题，否则你将不能利用数据发现有趣的事情。本书的第6章和第7章在讨论批处理时，将介绍进行即席查询的方法。

1.5.7 最少维护

维护是开发人员的重负，它是需要保持系统平稳运行的工作。包括预测什么时候添加用于扩展的机器、保持进程启动并运行以及调试生产环境中的任何错误。

减少维护的一个重要途径是选择实现复杂性尽量低的组件，即依赖底层具有简单机制的组件。特别是分布式数据库，它往往有着非常复杂的内部结构。系统越复杂，出错的可能性就越大，你越需要了解有关系统调试和优化的知识。

通过依靠简单的算法和组件，来降低实现复杂性，进而实现最少维护。Lambda 架构采用的一个技巧是：将复杂性推出核心组件，并将其送到几小时后输出是可丢弃的系统的片段中。所使用的最复杂的组件，如读/写分布式数据库，在输出最终是可丢弃的这一层中。本书第12章在讨论速度层时将详细讨论该技术。

1.5.8 可调试性

一旦出错，大数据系统必须提供必要的信息来调试系统。关键是能够追踪系统中的每一个值，并能明确该值是如何产生的。

“可调试性”是在 Lambda 架构中通过批处理层的功能特性和尽可能使用重新计算算法来实现的。

将所有这些属性组合在同一个系统中实现，这看起来是一个令人生畏的挑战。但从基本原理出发，正如 Lambda 架构做的，这些属性将从最终的系统设计中自然而然地生成。

在深入介绍 Lambda 架构之前，我们来看更传统的架构——以依赖增量计算为特征——以及它们无法满足这些属性的原因。

1.6 全增量架构的问题

在最高的层次上，传统的架构如图 1-3 所示。这种架构的特征是读 / 写数据库的使用以及随着新数据的可用，增量地维护这些数据库中数据的状态。例如，一个计算页面浏览量的增量方法，将通过对 URL 计数器加 1 来处理新的页面浏览量。这种架构的特征比关系型与非关系型更基础——事实上，



图 1-3 全增量架构

几十年来，绝大多数的关系型和非关系型数据库都是用全增量架构来部署的。

值得强调的是，全增量架构应用得如此广泛，以至于许多人没有意识到可以使用另外一种不同的架构来避免它们的问题。这是熟悉的复杂性的典型示例——复杂性如此根深蒂固，以至于人们认为无法避免它。

研究全增量架构的问题意义重大。通过查看任何全增量架构带来的一般复杂性，我们将开始这个主题的探索。然后，我们将查看针对相同问题的两种截然不同的解决方案：一种是使用最好的全增量解决方案；另一种是使用 Lambda 架构的解决方案。你会发现，全增量的版本在各方面明显更加糟糕。

1.6.1 操作复杂性

在全增量架构中有许多内在复杂性，给操作生产基础架构造成了困难。这里我们将关注其中一个方面——需要读 / 写数据库来执行在线合并，以及为保证这项工作平稳运行所必须做的操作。

在读 / 写数据库中，随着磁盘索引会逐步增加和修改，但部分索引从未使用过。这些未使用的部分索引占用空间，最终需要被回收以防磁盘被填满。如果索引一旦变成未使用的，回收空间就立刻回收，那样付出的代价太高，所以空间在一个被称为合并的进程中不定期地被批量予以回收。

合并是一类集中操作。在合并过程中，服务器对 CPU 和磁盘有更高的要求，这大大降低了该时期机器的性能。众所周知，数据库（如 HBase 和 Cassandra）都需要仔细地配置和管理，以免在合并时出现问题或服务器锁定。合并时性能的损失是一类甚至会导致级联故障的复杂性——如果太多机器同时执行合并操作，那么它们所支撑的负载将必须由集群中的其他机器处理。这可能使剩余的集群过载，导致彻底失败。这种失败已经出现过很多次了。

为了正确地管理合并，你必须在每个节点上都安排合并，以保证同一时间不会有太多节点受到影响；你必须知道一个合并操作会花多少时间以及时间的方差，以免有比预期更多的

节点在执行合并；你必须确保节点上有足够的磁盘容量，以保证在合并期间它们能够维持正常的操作；你还必须确保集群有足够的容量，以保证在合并时资源丢失就不会造成过载。

所有这些都可以由一个合格的操作人员来管理，但我们的观点是，处理任何一种复杂性的最好方式是完全摆脱这种复杂性。系统失败模式越少，就越不可能遇到意外的故障时间。处理在线合并是全增量架构中固有的复杂性，但在 Lambda 架构中，主数据库不需要任何在线合并。

1.6.2 实现最终一致性的极端复杂性

试图让系统高可用会导致增量架构的复杂性。高可用系统甚至允许在机器或部分网络发生故障时进行查询和更新。

事实证明，实现高可用性将与另一个称为一致性的属性直接竞争。具备一致性的系统返回结果时，会考虑以前所有的写操作。CAP (Consistency、Availability、Partition tolerance) 原理表明，在网络分区情况下，不可能在同一系统中同时实现高可用性和一致性，所以在一个网络分区中，高可用系统有时会返回陈旧的结果。

本书将在第 12 章深入讨论 CAP 原理——现在我们总是希望专注于不能实现的完全一致性和高可用性上，这会严重影响构建系统的能力。事实证明，如果业务需求中对高可用性的需要超过完全一致性，那么你必须处理大量的复杂性。

一旦网络分区结束，为了高可用性系统能回到一致性状态（即最终一致性），应用程序需要许多帮助，例如，在数据库中维护一个计数的基本用例，最显而易见的方法是在数据库中存储一个数值，当接收到需要加和的事件时，就做增量。你也许会很惊讶地发现，如果采取这种方法，在网络分区时，你会遇到海量数据丢失的情况。

导致这种情况的原因是，分布式数据库通过保存所有被存储信息的多个副本来实现高可用性。当你保存了相同信息的多份副本时，即使机器出现故障或网络分区，这些信息仍然可用，如图 1-4 所示。在网络分区时，选择成为高可用性的系统，只要副本是可获得的，就会有客户端的更新。这将导致副本产生分歧并接收不同的更新。只有分区消失，副本才可以合并成一个共同的值。

当网络分区开始时，假设有两个副本的计数为 10。假设第一个副本得到两个增量，第二个副本得到一个增量。当这些副本合并在一起时，值分别为 12 和 11，合并后的值应该是多少？虽然正确的答案是 13，但是没有办法通过查看数值 12 和 11 得到该值。这两个数可能在 11 的时候产生分歧（在这种情况下，答案会是 12），或者它们可能会在 0 的时候产生分歧（在这种情况下，答案是 23）。

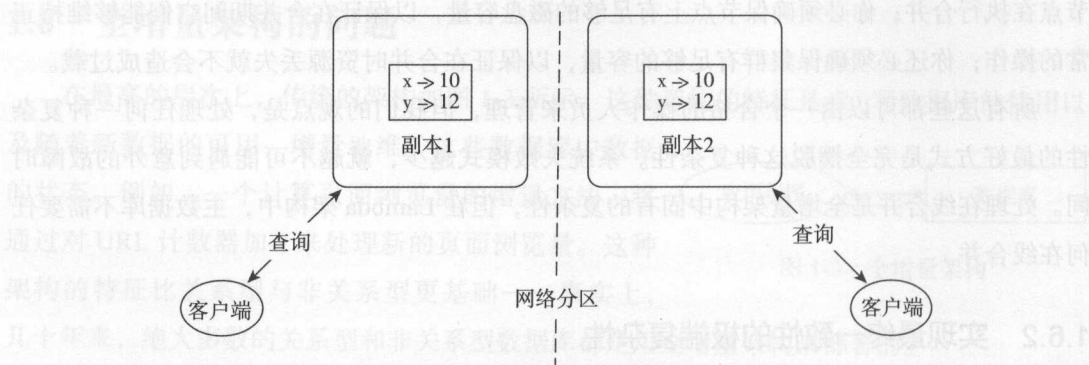


图 1-4 使用副本增加可用性

为了做高可用性的、正确的计算，只存储一个计数是不够的。当值出现分歧时，你需要一个负责合并的数据结构，并且需要实现一段用于分区结束后对值进行修复的代码。为了维护一个简单的计数，你必须处理令人难以置信的复杂性。

一般来说，在增量、高可用性系统中处理最终一致性，是不直观的且容易出错的。在高可用、全增量系统中，这种复杂性是固有的。稍后你将看到 Lambda 架构本身是如何以一种不同的方式，极大地减少实现高可用性、最终一致性系统的负担的。

1.6.3 缺乏容忍人为错误

我们希望指出的全增量架构的最后一个问题是，它们天生缺乏容忍人为错误的特性。增量系统不断修改保存在数据库中的状态，这意味着即使是一个错误也可以修改数据库中的状态。因为错误是不可避免的，所以全增量架构的数据库肯定会受到破坏。

重要的是要注意，全增量架构中，这是不用重新思考架构就可以解决的少数复杂性之一。下面考虑如图 1-5 所示的两种架构：同步架构，应用程序直接更新数据库；异步架构，在后台更新数据库之前事件先进到一个队列中。在这两种情况下，每个事件都被永久地记录到事件的数据存储中。通过保存每个事件，如果是人为错误导致数据库被破坏，那么你可以返回到事件存储，为数据库重建正确的状态。因为事件存储是不可变且不断增长的，冗余校验，如权限，可以放入事件存储，这使得不可能因出现某个错误而影响事件存储。这种技术也是 Lambda 架构的核心，我们将在第 2 章和第 3 章深入讨论。

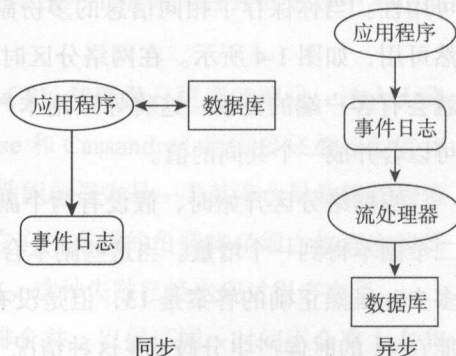


图 1-5 为全增量架构添加记录

尽管附带日志记录的全增量架构可以解决无日志记录的完全增量架构对人为错误缺乏容忍的缺陷，但是日志记录对于前面讨论的其他复杂性于事无补。在下一节中你将看到，纯粹基于完全增量计算的各种架构，包括那些附带日志记录的架构，需要努力解决很多问题。

1.6.4 全增量架构解决方案与 Lambda 架构解决方案

贯穿整本书实现的示例查询之一，适合作为全增量架构和 Lambda 架构的一个很好的对比。这个查询没有任何矫揉造作的地方——事实上，它是基于我们职业生涯中多次面临的现实生活中的问题的。这个查询用于处理网页浏览分析，并且完成对传入的两类数据的查询：

- ❑ 页面访问数据，包括用户 ID、URL 和时间戳。

- ❑ 等价数据，其中包含两个用户 ID。一份等价数据表明两个用户 ID 是指同一个人。

例如，你可能在电子邮箱 sally@gmail.com 和用户名 sally 之间有一份等价数据。如果 sally@gmail.com 也注册了用户名 sally2，那么你在 sally@gmail.com 和 sally2 之间有一份等价数据。通过传递性，你会知道用户名 sally 和 sally2 指的是同一个人。

查询的目的是计算一段时间内对一个 URL 来说独立访客的数量。查询应该将所有这段时间的数据加起来，并以最小的延迟（少于 100ms）来响应。查询的接口如下：

```
long uniquesOverTime(String url, int startHour, int endHour)
```

使得这个查询实现起来有些棘手的就是那些等价数据。如果在一个时间范围内，一个人用两个用户 ID 访问了相同的 URL，通过等价数据的连接（甚至传递），这应该只算一次访问。一个新的等价数据的传入，可以改变任何 URL 在任何时间范围内任何查询的结果。

在这一点上，我们将不去展示解决方案的细节，因为必须提及非常多的概念（如索引、分布式数据库、批处理、HyperLogLog 等）才能理解它们。此时让读者淹没在这些概念中往往会适得其反。相反，我们将专注于解决方案的特征和它们之间的显著差异。最佳完全增量解决方案将在第 10 章详细讨论，Lambda 架构解决方案将在第 8、9、14 和 15 章中进行讨论。

两种解决方案可以在准确性、延迟和吞吐量三个轴上进行对比。Lambda 架构解决方案在各方面表现得更胜一筹。这两种方案都必须实现近似值，但全增量版本被迫使用具有 3~5 倍甚至更糟糕的错误率的劣质近似技术。在全增量版本中执行查询的代价更高，并且会影响延迟和吞吐量。但这两种方案之间最显著的区别是：全增量版本需要使用特殊的硬件来实现接近合理的吞吐量。因为全增量版本必须做许多随机访问查找来解决查询问题，它实际上需要使用固态硬盘，以防磁盘寻道时间成为瓶颈。

Lambda 架构可以生成在每个方面都具有更高性能的解决方案，同时也避免了困扰全增量架构的复杂性，展示了正在发生的非常根本的事情——关键是挣脱全增量计算的束缚，采用不同的技术。现在让我们看看 Lambda 架构是如何做到这一点的。

1.7 Lambda 架构

实时计算任意数据集上的任意函数，是一个令人望而却步的问题。没有单独的工具可以提供完整的解决方案，相反，你必须使用各种工具和技术构建一个完整的大数据系统。

Lambda 架构的主要思想是将大数据系统建立为一系列的层，如图 1-6 所示。每一层满足属性的一个子集，且通过该层的下一层所提供的功能来构建。虽然你通过整本书来学习如何设计、实现和部署每一层，但整个系统是如何有机结合在一起的高层思想是很容易理解的。

一切从 `query = function(all data)` 等式开始。理想情况下，你可以不断运行这个函数来获取结果。不幸的是，即使这是可能的，也需要占用大量的资源，并且相当昂贵。想象一下，每次你想要响应某人当前位置的查询时，都必须读取 1PB 的数据集。

最显而易见的替代方法是预先计算查询函数。我们将预先计算的查询函数称为**批处理视图**（Batch View）。这不是动态计算查询，而是从预先计算好的视图中读取结果。预先计算的视图是有索引的，因此可以用随机读取的方式进行访问。该系统看起来像是这样的：

```
batch view = function(all data)
query = function(batch view)
```

在该系统中，你在所有数据上运行一个函数来获取批处理视图。然后，当你想知道一个查询的值时，只需针对批处理视图运行一个函数即可。批处理视图可以使你很快地从中获得所需要的值，而无须扫描所有数据。

这个讨论有点抽象，下面举例说明。假设你正在构建一个网络分析的应用程序（再次构建），并且想要查询任一范围天数内某个 URL 的页面浏览量。如果你在所有数据上运行该查询函数，最好扫描数据集中相应时间范围内那个 URL 的页面浏览量，并返回结果的计数。

批处理视图方法替代在所有页面浏览上运行一个函数的方式，来预先计算 `[url,day]` 键的索引，以获得某天某个 URL 页面浏览的计数。然后为了解决该查询，从视图中检索相应时间范围内的所有天的值，并将所有计数相加，以得到最终的结果。这种方法如图 1-7 所示。

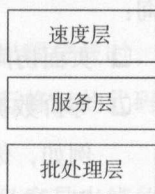


图 1-6 Lambda 架构

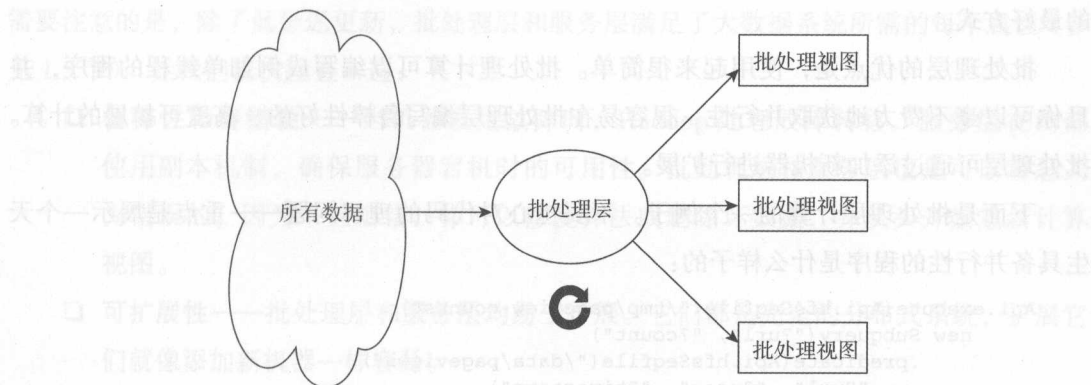


图 1-7 批处理层的架构

应该明确的是，到目前为止所描述的这种方法似乎缺了点什么。创建批处理视图显然会是一个高延迟操作，因为它是在你的所有数据上运行一个函数。在批处理视图结束时，很多新的数据将被收集但是没有展示在批处理视图中，并且这个查询将过期许多小时。但是我们暂时忽略这个问题，因为它是可以解决的。假设过期几个小时的查询是可以的，那么我们继续探索通过在完整数据集上运行函数来预先计算批处理视图的这个想法。

1.7.1 批处理层

Lambda 架构中实现 `batch view = function(alldata)` 等式的这部分被称为**批处理层**。批处理层存储数据集的主副本，并在主数据集上预先计算批处理视图（见图 1-8）。主数据集可以被视作一个非常大的记录列表。

批处理层需要能够做两件事：存储不可变的、不断增长的主数据集；在该数据集上运行任意函数。最好使用批处理系统完成这种类型的处理。Hadoop 是批处理系统的一个典型例子，我们将在本书中使用 Hadoop 来阐述批处理层的概念。

批处理层最简单的形式可以用如下的伪代码表示：

```
function runBatchLayer():
  while(true):
    recomputeBatchViews()
```

批处理层在 `while(true)` 中循环运行，不断从头开始重新计算批处理视图。实际上，批处理层的功能并不仅限于此，相关内容会在本书后续章节予以介绍。这里只探讨批处理层

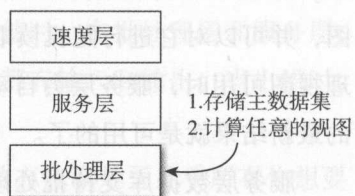


图 1-8 批处理层

的最好方式。

批处理层的优点是，使用起来很简单。批处理计算可以编写成例如单线程的程序，并且你可以毫不费力地获取并行性。很容易在批处理层编写鲁棒性好的、高度可扩展的计算。批处理层可通过添加新机器进行扩展。

下面是批处理层计算的一个例子。不要担心对代码的理解问题——重点是展示一个天生具备并行性的程序是什么样子的：

```
Api.execute(Api.hfsSeqfile("/tmp/pageview-counts"),
    new Subquery("?url", "?count")
        .predicate(Api.hfsSeqfile("/data/pageviews"),
            "?url", "?user", "?timestamp")
        .predicate(new Count(), "?count"));
```

这段代码把给定的原始页面浏览的数据集作为输入，为每个 URL 计算页面浏览量。这段代码的有趣之处在于——调度工作的所有并行性挑战和结果的合并都已经为你做好了。由于算法是用这种方式写的，因此它可以任意地分布在 MapReduce 集群中，扩展到可用的不管多少数量的节点上。在计算结束时，输出目录将包含一些结果文件。本书第 7 章将介绍如何编写这样的程序。

1.7.2 服务层

批处理层的功能是生成批处理视图。下一步是在某个地方加载视图，以便它们可以被查询到——这个地方就是服务层。服务层（Serving Layer）是一个专用的分布式数据库，用于加载批处理视图，并可以对它进行随机读取（见图 1-9）。当新的批处理视图可用时，服务层会自动替换那些视图，这样更多的最新结果就是可用的了。

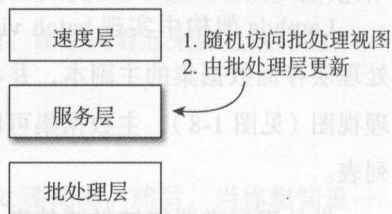


图 1-9 服务层

服务层数据库支持批处理更新和随机读取。最值得注意的是，它不需要支持随机写操作——这是非常重要的一点，因为随机写会在数据库中导致绝大多数的复杂性——因为不支持随机写，所以这些数据库非常简单。简单性使得它们是可预测的、鲁棒性好的，易于配置，且操作简单。ElephantDB 是你将在本书中学习使用的服务层数据库，它只有几千行代码。

1.7.3 批处理层和服务层满足几乎所有属性

批处理层和服务层支持对任意数据集上的任意查询，当然是以查询将过期几个小时为代价的。一个新的数据片段从批处理层传播到可以被查询到的服务层，需要花费几个小时。

需要注意的是，除了低延迟更新，批处理层和服务层满足了大数据系统所需的每个属性（参见 1.5 节）。让我们依次检查一遍：

- **鲁棒性和容错性**——当机器发生故障时，Hadoop 处理故障转移。服务层在内部使用副本机制，确保服务器宕机时的可用性。批处理层和服务层也是可以容忍人为错误的，因为一旦出错，你可以修复算法或删除坏数据，并从头开始重新计算视图。
- **可扩展性**——批处理层和服务层均易于扩展。它们都是完全的分布式系统，扩展它们就像添加新机器一样容易。
- **通用性**——这里描述的架构非常通用。你可以计算和更新任意数据集的任意视图。
- **延展性**——添加一个新的视图就像对主数据集添加一个新函数一样容易。由于主数据集可以包含任意数据，因此新类型的数据可以很容易地被添加进来。如果想微调一个视图，你不必担心支持应用程序中多个版本的视图。你可以简单地从头开始重新计算整个视图。
- **即席查询**——批处理层支持即席查询。在某个位置，所有数据都是方便可用的。
- **最少维护**——维护这个系统的主要组件是 Hadoop。Hadoop 需要一些管理知识，但操作起来相当简单。正如前面所解释的那样，服务层数据库是很简单的，因为它们不进行随机写操作。因为服务层数据库有很少的活动部件，出错的可能性不大。因此，服务层数据库不太可能出错，从而更容易维护。
- **可调试性**——在批处理层运行时，你永远都有计算的输入和输出。在传统数据库中，一个输出可以替换原来的输入——比如递增一个值时。在批处理层和服务层，输入是主数据集，输出是视图。同样，所有中间步骤都有输入和输出。当出错时，输入和输出可以给出调试所需要的所有信息。

批处理层和服务层之美在于它们用一个简单且易于理解的方法满足了几乎所有你想要的属性——没有并发问题要处理，扩展非常简单。唯一没有满足的属性是低延迟更新。在最后一层，速度层，会解决这个问题。

1.7.4 速度层

一旦批处理层完成预先计算批处理视图，服务层即进行更新。这意味着唯一没有在批处理视图中展示的数据是运行预先计算期间新来的数据。为了实现一个完全的实时数据系统，剩下的任务要完成——也就是说，为了实时在任意数据上执行任意函数计算——弥补最后几个小时的那些数据。这是速度层的目的。顾名思义，它的目标是确保

新的数据按照应用程序的需求尽快展示在查询函数中（见图 1-10）。

你可以认为速度层是类似于批处理层的，它基于接收到的数据生成视图。两者之间一个很大的区别是，速度层只查看最近的数据，而批处理层要立即查看所有数据。另一个很大的区别是，为达到最小延迟的可能，速度层不会立即查看所有新数据。相反，每当接收到新的数据，它就更新实时视图，而不是像

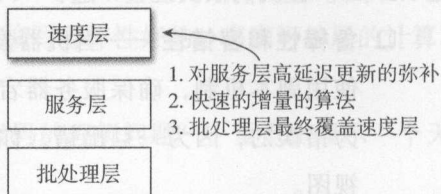


图 1-10 速度层

批处理层从头开始重新计算视图。速度层做增量计算，而不是像批处理层那样进行重新计算。

我们可以将速度层上的数据流格式化成以下等式：

$$\text{realtime view} = \text{function}(\text{realtime view}, \text{new data})$$

实时视图基于新数据和现有的实时视图进行更新。

Lambda 架构总体被总结为以下三个等式：

$$\text{batch view} = \text{function}(\text{all data})$$

$$\text{realtime view} = \text{function}(\text{realtime view}, \text{new data})$$

$$\text{query} = \text{function}(\text{batch view}, \text{realtime view})$$

这些想法的图解如图 1-11 所示。也就是说，不是只需要对批处理视图运行一个函数就可以得到查询，而是需要通过查看批处理视图和实时视图并将结果合并在一起，才能得到查询。

速度层使用支持随机读取和随机写入的数据库。因为这些数据库支持随机写，所以它们比在服务层使用的数据库要高出几个数量级的复杂性，无论是实现方面还是操作方面。

Lambda 架构之美在于，一旦数据通过批处理层到服务层，实时视图中相应的结果就不再需要了。这意味着你可以丢弃不再需要的实时视图。这是一个很好的结果，因为速度层远比批处理层和服务层更复杂。Lambda 架构的这个属性被称为**复杂性隔离**，这意味着复杂性被推入一个只存储暂时结果的层中。如果有什么差错，你可以丢弃整个速度层的状态，并且在几小时内使一切恢复正常。

下面继续构建网络分析应用程序的例子——它支持一些天当中页面浏览量的查询。请回顾一下批处理层从 [url, day] 到页面浏览量所生成的批处理视图。

速度层保存自己独立的 [url, day] 到页面浏览量的视图。而批处理层通过逐次计算页面浏览，重新计算批处理视图。每当接收到新数据，速度层就通过增加在视图中的计数值来更新自己的视图。为解决一个查询过程，你需要查询所需的批处理视图和实时视图，来满足指定的日期范围，并将结果相加以得到最终计数。还有一项需要去做的工作，就是恰当地同步结果（相应内容将在本书后续的章节中介绍）。

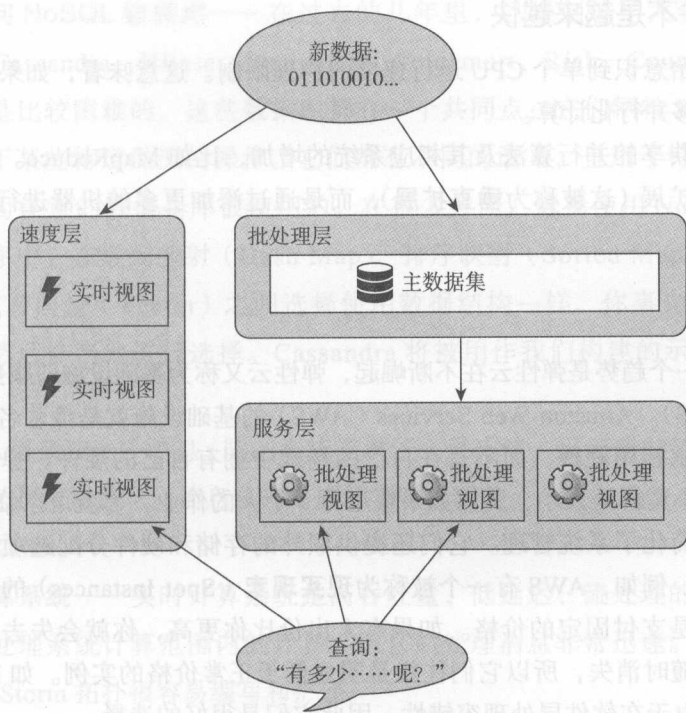


图 1-11 Lambda 架构图解

一些算法很难增量地计算。批处理 / 速度层的分离，为你在批处理层上使用精确算法和速度层上使用近似算法提供了足够的灵活性。批处理层多次重写速度层，所以近似值得到修正，并且系统也显示了**最终准确性**的属性。例如计算独立计数，如果独立值的集合很大，这可能是很有挑战性的。很容易在批处理层完成独立值计数，因为你能立即看到所有数据，但在速度层你可以将 HyperLogLog 集合作为一个近似值使用。

最终，性能和鲁棒性兼得。因为批处理层纠正在速度层中的计算，所以在批处理层做准确计算且在速度层做近似计算的系统，呈现了最终的准确性。你得到的仍然是低延迟更新，但由于速度层是暂时的，实现该属性的复杂性并不会影响结果的鲁棒性。当涉及性能折中方案时，速度层的暂时特性给你提供了极大的灵活性。当然，因为用增量方式完成的计算是准确的，所以该系统是完全准确的。

1.8 技术上的最新趋势

理解本书中所使用的工具的背景是很有帮助的。技术上的众多趋势深深影响着构建大数据系统的方式。

1.8.1 CPU 并不是越来越快

人们已经开始意识到单个 CPU 运行速度的物理限制。这意味着，如果想扩展到更多的数据，你必须能够并行化计算。

这导致了无共享的并行算法及其相应系统的增加，比如 MapReduce。不是只通过购买更好的机器进行扩展（这被称为**垂直扩展**），而是通过添加更多的机器进行扩展（这被称为**水平扩展**）。

1.8.2 弹性云

技术上的另一个趋势是弹性云在不断崛起，弹性云又称为**基础设施即服务**（Infrastructure as a Service, IaaS）。Amazon Web Services（AWS）的基础设施就是最著名的弹性云。弹性云允许你根据需求租用硬件，而不是在自己的场地中拥有自己的硬件。弹性云几乎可以瞬间让你增加或减小集群的大小，所以如果要运行一个大的作业，你就可以临时地分配硬件。

弹性云大大简化了系统管理。它们还提供额外的存储和硬件分配选项，可以显著降低基础设施的价格。例如，AWS 有一个被称为**现买现卖**（Spot Instances）的特性，即你对实例进行投标而不是支付固定的价格。如果有人出价比你更高，你就会失去该实例。因为现买现卖特性可以随时消失，所以它们往往是明显低于正常价格的实例。如 MapReduce 的分布式计算系统，由于在软件层处理容错性，因此它们是很好的选择。

1.8.3 大数据充满活力的开源生态系统

在过去的几年里，开源社区创造了数量庞大的大数据技术。本书中所授的所有技术都是开源且免费使用的。

你将学习五类开源项目。记住，这不是一本调研书——其目的不是只教一堆技术。你将学习基本原则，以便能够评估和选择适合自身需求的工具。

- ❑ **批处理计算系统**——批处理计算系统是高吞吐量、高延迟的系统。批处理计算系统几乎可以做任意计算，但是它们可能需要几小时或几天。本书唯一使用的批处理计算系统是 Hadoop。Hadoop 项目有两个子项目：Hadoop 分布式文件系统（Hadoop Distributed File System, HDFS）和 Hadoop MapReduce。HDFS 是分布式的、容错的存储系统，可以扩展到 PB 级别的数据。MapReduce 是一个集成了 HDFS、水平可扩展的计算框架。
- ❑ **序列化框架**——序列化框架为不同语言间使用的对象提供了工具和库。它们可以将任何语言的对象序列化为一个字节数组，然后将字节数组反序列化成任何语言的对象。序列化框架提供了一种模式定义语言（Schema Definition Language）来定义对象和对象的字段，它们为安全地版本化对象提供了机制，这样不用使现有对象无效就可以形成模式。三个著名的序列化框架是 Thrift、Protocol Buffers 和 Avro。

❑ **随机访问 NoSQL 数据库**——在过去的几年里，已经创建了大量的 NoSQL 数据库。如 Cassandra、HBase、MongoDB、Voldemort、Riak、CouchDB 等，完全理解它们是比较困难的。这些数据库都有一个共同点：它们牺牲 SQL 的完整表达，而专注于某些特定类型的操作。它们都有不同的语义，且用于特定的目的。它们不是作为任意的数据仓库被使用的。在很多方面，选择使用 NoSQL 数据库，就像在程序中，在散列映射（Hash Map）、排序映射（Sorted Map）、链表（Linked List）或者向量（Vector）之间选择使用数据结构一样。你事先要知道自己想做什么，然后恰当地进行选择。Cassandra 将被用作我们构建的示例应用程序的一部分。

❑ **消息 / 队列系统**——消息 / 队列系统提供了一种方法，以容错和异步的方式在进程之间发送和使用消息。消息队列是进行实时处理的一个关键组件。本书将使用的是 ApacheKafka。

❑ **实时计算系统**——实时计算系统是高吞吐量、低延迟、流处理的系统。它们无法进行批处理系统计算范围内的计算，但它们处理消息非常迅速。本书将使用的是 Storm。Storm 拓扑很容易编写和扩展。

随着这些开源项目的成熟，与之相关的企业已经成形并能提供企业级的支持，例如，Cloudera 提供 Hadoop 支持，DataStax 提供 Cassandra 支持，其他项目都是公司产品，例如，Riak 是 Basho 科技的产品，MongoDB 是 10gen 的产品，RabbitMQ 是 SpringSource 的产品——它是 VMWare 的一个部门。

1.9 示例应用：SuperWebAnalytics.com

在本书中，我们将创建一个大数据应用程序示例来说明一些概念。我们将为 Google Analytics 构建数据管理层——比如服务。该服务将能够每天追踪数十亿的页面浏览量。该服务将支持多种不同的指标。每个指标都被实时地支持。指标的范围很广——从简单的统计指标，到访客是如何浏览网站的复杂分析指标。

示例应用将支持的指标如下：

- ❑ **按照时间切片基于 URL 的页面浏览计数**——示例查询是“过去一年中每一天的页面浏览量是多少？”和“过去 12 小时内有多少页面浏览量？”
- ❑ **按照时间切片基于 URL 的独立访客**——示例查询是“2010 年有多少独立访客访问这个域名？”和“过去三天内每个小时，有多少访客访问这个域名？”

❑ 跳出率分析——“用户访问该站点的某个页面，没有访问其他任何页面的百分比是多少？”

我们将构建存储、处理并为应用程序提供查询的层。

1.10 总结

你已经知道了使用传统技术（如分片）来扩展关系型系统时会出现的错误。而我们面临的问题不仅仅是扩展，因为那会使系统变得更难以管理、扩展，甚至更难理解。在后面的章节中，当介绍如何构建大数据系统时，我们将像注重可扩展性一样注重鲁棒性。你将看到，当用正确的方式构建系统时，鲁棒性和可扩展性都是可以在同一个系统中实现的。

使用 Lambda 架构构建数据系统的好处不仅仅是可扩展，因为你的系统将能够处理大量的数据、收集更多的数据并获得更多的价值。增加存储数据的数量和类型，将会有更多机会去挖掘数据、生成分析和构建新的应用程序。

使用 Lambda 架构的另一个好处是应用程序的健壮性很好。原因有很多，例如，你将有能力在整个数据集上运行计算来进行迁移或解决出错的事情。你永远不需要处理同一时间模式中有多个活跃版本的情况。当改变模式时，你将有能力将所有数据更新到新的模式。同样的，如果一个错误的算法被不小心部署到生产环境，破坏了所提供的数据库，你可以通过重新计算被破坏的数值很容易地解决该问题。如你将见到的，还有许多其他原因使得大数据应用程序的鲁棒性更好。

最后，性能将是更加可预测的。虽然 Lambda 架构作为一个整体是通用和灵活的，但组成系统的各个组件是特定的。当与诸如 SQL 的查询计划比较时，后台很少会有“魔法”发生。这就使我们可以得到更加可预测的性能。

如果你对很多这种信息仍然不确定，也不用担心。我们还有很多内容需要进行探讨，并且将通过本书的课程再次深入讨论本章中介绍的每一个主题。在下一章中，你将开始学习如何构建 Lambda 架构。你会从堆栈的核心开始，即如何对数据集的主副本进行建模和系统化。

第一部分 *Part I*

批处理层

第一部分主要介绍 Lambda 架构的批处理层。这部分的章节结合示例讲述相关理论。

第2章探讨了如何对主数据集中的数据进行建模和视图化。

第3章使用 Apache Thrift 工具来阐述第2章涉及的概念。

第4章探讨了主数据集的存储要求。你会发现数据库解决方案提供的许多典型特性不适用于主数据集，而且实际上阻碍了主数据存储的优化。一个更简单、更精致的全能型存储解决方案可以更好地满足需求。

第5章使用 Hadoop 的分布式文件系统来阐述主数据集的物理存储。

第6章探讨了在主数据集上使用 MapReduce 范式来计算任意的函数。一般来说，MapReduce 足以计算任何可扩展的函数。尽管 MapReduce 是强大的，但是你会发现更高阶的抽象应用程序会使它变得更容易使用。

第7章会介绍一个名为 JCascalog 的强大 MapReduce 高阶抽象应用。

为了将所有概念联系起来，第8章和第9章给出了为运行 SuperWebAnalytics.com 示例而实现的完整批处理层。其中，第8章介绍总体架构和算法，而第9章详细介绍工作代码。

.....

按照本书的侧重点，我们将围绕一个示例应用程序展开，但重要事实是我们在本章中讨论

大数据的数据模型

本章内容

- 数据的属性
- 基于事实的数据模型
- 大数据基于事实的模型的优点
- 图模式

在第1章中，你看到了使用传统工具构建数据系统时所走入的误区，然后我们回到基本原理，以获得更好的设计。可以看到，每一个数据系统都可以被定义为数据上用来计算的函数，并且你学习了 Lambda 架构的基础，这种架构提供了一种用来实时声明任意数据上的任意函数的可行性方法。

Lambda 架构的核心是主数据集，这在图 2-1 中是突出显示的。在 Lambda 架构中，主数据集是计算系统中的事实来源。即使失去了所有的服务层数据集和速度层数据集，你仍然可以从主数据集中重构应用程序。这是因为由服务层来服务的批处理视图是由主数据集上的函数生成的，并且由于速度层只基于近期的数据，它可以在几个小时内进行自动重构。

主数据集是 Lambda 架构中唯一不能受损的部分。过载的机器、失效的磁盘以及停电都可能导致错误的发生，动态数据系统中的人为错误是内在风险且可能无法避免。你必须仔细设计主数据集，以避免所有这些情况下的数据损坏，因为容错对于长时间运行的数据系统能够正常工作是至关重要的。

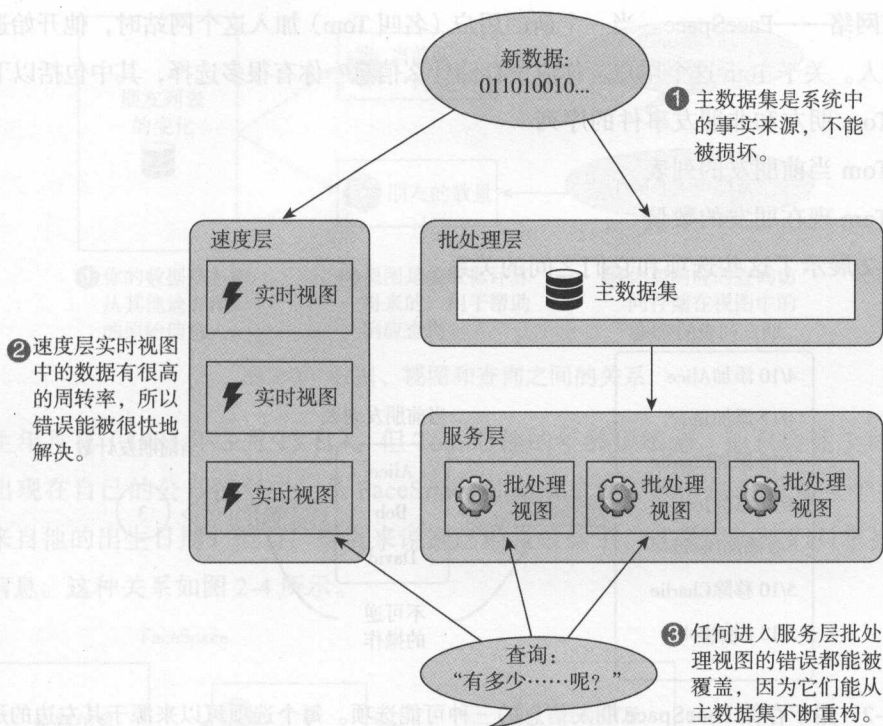


图 2-1 Lambda 架构中的主数据集作为大数据系统计算结果的来源。服务层和速度层的错误是可以纠正的, 但主数据集中的数据损坏是不可挽回的

对于主数据集, 我们将探讨两个方面: 所使用的数据模型以及如何物理地存储主数据集。本章介绍主数据集的数据模型的设计以及一个数据模型应有的属性。第 3 章介绍如何物理地存储主数据集。

在本章, 你应掌握如下内容:

- 掌握数据的关键属性
- 查看这些属性如何维护基于事实的模型
- 查看主数据集基于事实的模型的优点
- 使用图模式表示基于事实的模型

下面从具有普遍性的术语数据展开讨论。

2.1 数据的属性

按照本书的侧重点, 我们将围绕一个示例应用程序展开讨论。假设你正在设计下一个

大型社交网络——FaceSpace。当一个新的用户（名叫 Tom）加入这个网站时，他开始邀请他的朋友和家人。关于 Tom 这个用户，你应该存储什么信息？你有很多选择，其中包括以下几种：

- ❑ Tom 朋友和非朋友事件的序列
- ❑ Tom 当前朋友的列表
- ❑ Tom 现在朋友的数量

图 2-2 展示了这些选项和它们之间的关系。

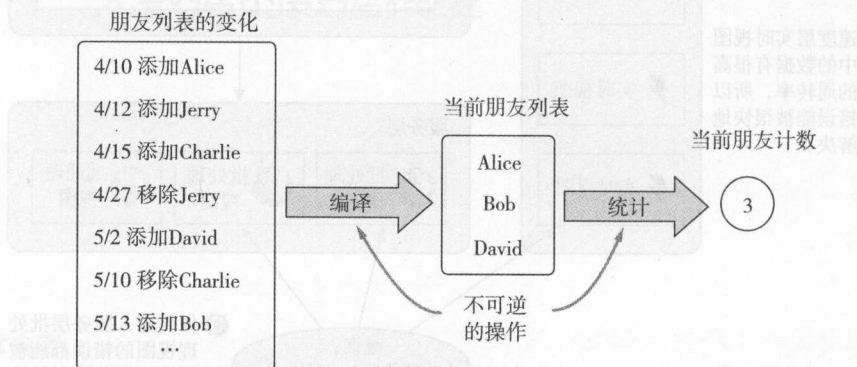


图 2-2 用于存储 FaceSpace 朋友信息的三种可能选项。每个选项可以来源于其左边的那个选项，但这是一个单向的过程

这个例子说明了信息的依赖性。请注意，每一层的信息可以从前一个（其左边）获取到，但这是一个单向的过程。从朋友和非朋友事件的序列，你可以确定其他的数量。但是如果你只有朋友的数量，就不可能准确地知道他们是谁。同样，从当前朋友的列表中，不可能确定 Tom 是 Jerry 以前的一个朋友，或者近来 Tom 的社交圈子有所增长。

概念的依赖性决定了我们将要使用的术语定义：

- ❑ **信息**是与大数据系统相关的知识的一般集合。这是“数据”一词的口语化表达。
- ❑ **数据**是指不能来源于任何其他东西的信息。数据是衍生一切的公理。
- ❑ **查询**是向数据“询问”的问题。例如，你通过查询财务交易历史来确定自己目前的银行账户余额。
- ❑ **视图**是来自基础数据的信息，用来协助完成特定类型的查询。

图 2-3 展示了 FaceSpace 的数据、视图和查询这些术语的信息依赖性。

注意，一个人的数据可以成为另一个人的视图，这是很重要的。假设 FaceSpace 已成为一个非常成功的网站，并且现有某第三方广告商创建了一个网络爬虫，用来从该网站的用户资料中获取人口统计信息。FaceSpace 可以完全访问 Tom 所提供的信息——比如他完

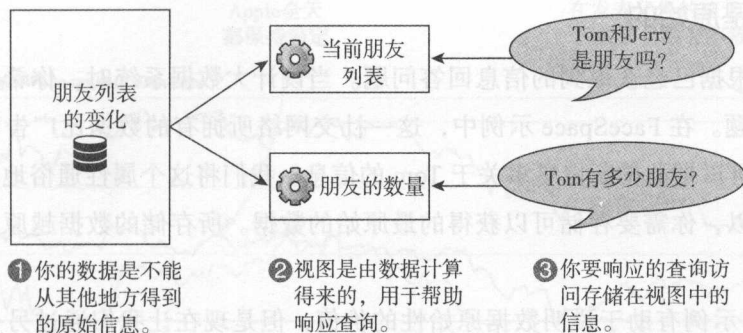


图 2-3 数据、视图和查询之间的关系

整的出生年月日（1984 年 3 月 13 日）。但 Tom 对他的年龄很敏感，他只会让 3 月 13 日这一信息出现在自己的公开资料中。从 FaceSpace 的角度来看，Tom 的生日是一个视图，因为生日来自他的出生日期，但对广告商来说这已经是数据了，这是因为他们只掌握 Tom 很有限的信息。这种关系如图 2-4 所示。

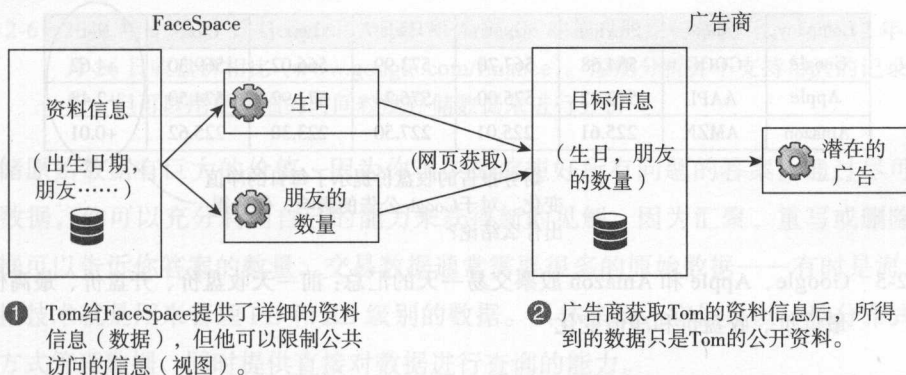


图 2-4 根据你看待数据的角度，将信息分类为数据或视图。对于 FaceSpace，Tom 的生日是一个视图，因为生日来自他的出生日期，但是第三方广告商将此视作数据

通过建立一个共享的词汇库，我们现在可以介绍数据的关键属性：**原始性**、**不变性**和**永久性**（或“数据的永恒真实性”）。这三个关键属性是理解大数据系统的基础。

如果你有关系型数据库的知识背景，这可能会让你感到困惑。典型的表现就是你会通过不断更新和总结信息来反映当前世界的状态，而不关心不变性或永久性。但这种方法限制了你可以利用数据回答的问题，并且对于避免错误和损坏来说也不够“鲁棒”。通过实现大数据世界里的关键属性，你会构建一个鲁棒性更高的系统，获得更强大的功能。

下面将深入讨论数据的原始性。

2.1.1 数据是原始的

数据系统根据已经获取到的信息回答问题。当设计大数据系统时，你希望它能够回答尽可能多的问题。在 FaceSpace 示例中，这一社交网络所拥有的数据比广告商的数据更有价值，因为你可以据此推断出更多关于 Tom 的信息。我们将这个属性通俗地称为数据的**原始性**。如果可以，你需要存储可以获得的最原始的数据。所存储的数据越原始，可以问的问题就越多。

FaceSpace 示例有助于说明数据原始性的价值，但是现在让我们通过另一个例子来更好地理解这个观点。股市交易价格是信息的来源，每天都有数以百万计的股票和数十亿的美元在转手。随着这些海量交易的发生，我们每天都会记录股票的历史价格，包括开盘价、最高价、最低价和收盘价。但这些数据通常无法描述股市的大局，并且可能会改变你对某些事情的看法。如图 2-5 所示，它记录了当 Google 针对竞争对手发布了新产品时，Google、Apple 和 Amazon 的股票在一天之内的价格数据。

Company	Symbol	Previous	Open	High	Low	Close	Net
Google	GOOG	564.68	567.70	573.99	566.02	569.30	+4.62
Apple	AAPL	572.02	575.00	576.74	571.92	574.50	+2.48
Amazon	AMZN	225.61	225.01	227.50	223.30	225.62	+0.01

财务报告的收盘价提示了每日的净值变化。对于 Google 公告的影响，你会得出什么结论？

图 2-5 Google、Apple 和 Amazon 股票交易一天的汇总：前一天收盘价、开盘价、最高价、最低价、收盘价和净值变化

这份数据表明，Amazon 可能没有受到 Google 公告的影响，因为它的股票价格只有轻微波动。这份数据还表明，Google 的公告对 Apple 完全没有影响，或者说造成了积极的影响。

但是如果可以访问以更细时间粒度存储的数据，你就能得到关于那一天所发生事件更清晰的总体图像，并据此进一步探究潜在的因果关系。图 2-6 实时描述了三家公司股票价格的相对变化，这表明 Amazon 和 Apple 确实受到了这份公告的影响，且 Amazon 比 Apple 受到的影响更大。

还要注意的，额外的数据可以阐述一些你可能在查看原来每天的股票价格概要时没有考虑到的新方向。例如，更细粒度的数据让你怀疑 Amazon 是否受到了更大的影响，因为 Google 的新产品与 Amazon 在平板电脑和云计算市场都产生了竞争。

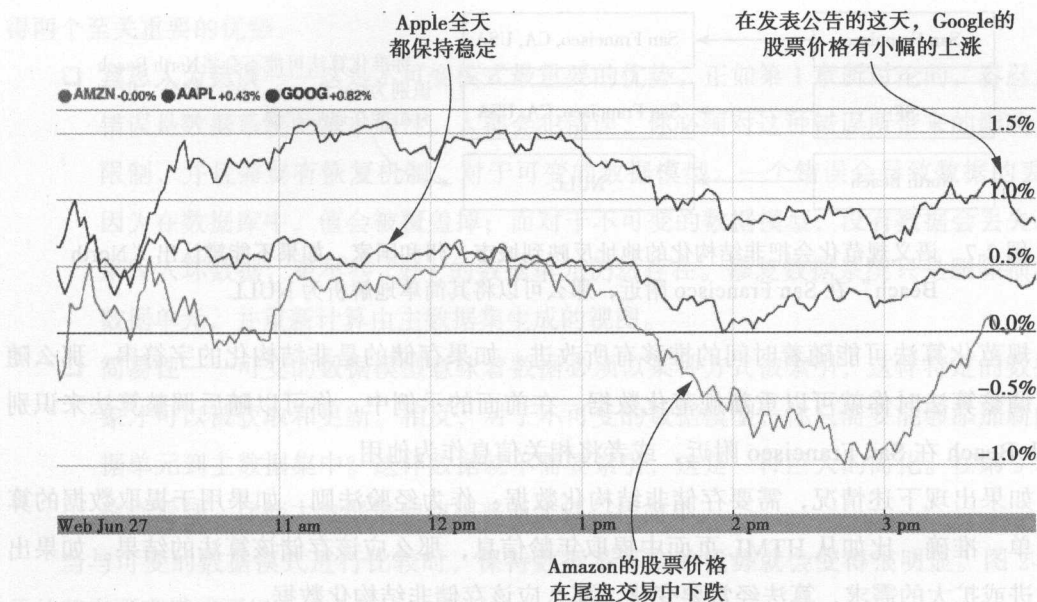


图 2-6 2012 年 6 月 27 日 Google、Apple 和 Amazon 的相对股票价格变化，与 2012 年 6 月 26 日收盘价相比 (www.google.com/finance)。短期分析并不支持每天的记录分析，但可以用更精细的时间粒度存储数据来进行分析

存储原始数据有巨大的价值，因为你很少提前想好所有问题的答案。通过尽可能地保存原始数据，你可以充分利用自己的能力来获得新的见解，因为汇聚、重写或删除信息限制了数据可以告诉你答案的数量。交易数据通常需要很多的原始数据——有时是海量数据。而大数据技术就是用来管理 PB 和 EB 级别的数据。具体来说，它们是以一种分布式的、可扩展的方式管理数据，同时提供直接对数据进行查询的能力。

尽管概念很简单，但你并不总是清楚应该存储什么信息以作为原始数据。下面将提供几个例子来帮助和引导你做出这个决定。

1. 非结构化的数据比规范化的数据更原始

在决定存储什么原始数据时，一个常见的令人困惑之处就是解析和语义规范化之间的界线。语义规范化是将任意格式的信息变成结构化形式的过程。

例如，FaceSpace 可能会请求 Tom 给出自己所处的地理位置。他可能会输入该字段的任何信息，如 San Francisco (旧金山)、CA、SF、North Beach (北滩) 等。语义规范化算法会尝试将输入数据匹配为已知的地方，如图 2-7 所示。

如果遇到非结构化位置的字符串形式的数据，那么随后在调整算法时你应该存储非结构化的字符串形式，还是语义规范化的形式？我们认为最好存储非结构化的字符串，因为

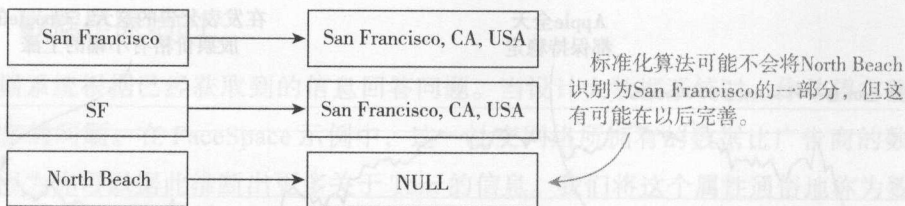


图 2-7 语义规范化会把非结构化的地址反映到城市、州和国家。如果不能辨认出“North Beach”在 San Francisco 附近，那么可以将其简单地解析为 NULL

语义规范化算法可能随着时间的推移有所改进。如果存储的是非结构化的字符串，那么随后在调整算法时你就可以重新规范化数据。在前面的示例中，你可以随后调整算法来识别 North Beach 在 San Francisco 附近，或者将相关信息作为他用。

如果出现下述情况，需要存储非结构化数据：作为经验法则，如果用于提取数据的算法简单、准确，比如从 HTML 页面中提取年龄信息，那么应该存储该算法的结果。如果出于改进或扩大的需求，算法经常要变化，那么应该存储非结构化数据。

2. 更多的信息并不意味着更原始的数据

通常人们会很容易地认为，更多的数据相当于更原始的数据，但事实并非总是如此。假设 Tom 是一个博主，他想将自己的文章添加到 FaceSpace 资料中。当 Tom 提供了他博客的 URL 地址时，你究竟应该存储什么？

存储博客记录的纯文本无疑是一种可能性。但 Tom 特意强调了任何斜体、黑体、大字体的短语——在文本分析中，它们可能是有用的。例如，你可以使用这些额外的信息作为索引，使得 FaceSpace 可以被搜索到。因此我们认为，与 ASCII 的文本字符串相比，带注释的文本记录是一种形式更为原始的数据。

此外，你也可以存储 Tom 博客完整的 HTML。然而，这会在总字节数、配色方案、网站样式方面存储相当多的数据，并且该网址的 JavaScript 代码不能用来获得任何关于 Tom 的额外信息。它们只是作为网站内容的容器，而不应该作为原始数据的一部分。

2.1.2 数据是不可变的

如果你精通关系型数据库，想必会觉得不可变的数据似乎是一个奇怪的概念。毕竟在关系型数据库和大多数其他数据库中，更新是最基本的操作之一。但就不变性来说，你不会更新或删除数据，你只是添加更多的数据[⊖]。通过对大数据系统使用不可变模式，你将获

⊖ 有几个场景你可以删除数据，但这些都是特殊情况，不属于系统日常工作流的一部分。我们将在 2.1.3 节中讨论这些场景。

得两个至关重要的优势：

❑ **容忍人为错误**——这是不可变模式最重要的优势。正如第1章所讨论的，容忍人为错误是数据系统的基本属性。人都会犯错误，你必须对这种错误所带来的影响予以限制，并且需要有恢复机制。对于可变的数据模型，一个错误会导致数据的丢失，因为在数据库中，值会被覆盖掉；而对于不可变的数据模型，没有数据会丢失。如果写入坏数据，更早些（好）的数据单元仍然存在。修复数据系统只是删除损坏的数据单元，并重新计算由主数据集生成的视图。

❑ **简易性**——可变的数据模型意味着数据必须以某种方式被索引，这样特定的数据对象才可以被获取和更新。相反，对于不可变的数据模型，你只需要能够添加新的数据单元到主数据集中。这样数据就不需要索引，这是一种巨大的简化。在第3章中你会看到，存储主数据集和使用普通文件一样简单。

当与可变的数据模式进行比较时，保持数据不可变的优势就会变得很明显。图2-8所示的基本可变模式可以为 FaceSpace 所用。

用户信息					
ID	名称	年龄	性别	公司	位置
1	Alice	25	女	Apple	Atlanta, GA
2	Bob	36	男	SAS	Chicago, IL
3	Tom	28	男	Google	San Francisco, CA
4	Charlie	25	男	Microsoft	Washington, DC
...

如果Tom搬到不同的城市，该值会被覆盖。

图 2-8 FaceSpace 用户信息的可变模型。当细节发生改变——比如 Tom 所处的位置变化到 Los Angeles——以前位置的值会被覆盖并丢失

如果 Tom 搬到 Los Angeles（洛杉矶），你就要更新突出显示的记录，以反映他目前所处的位置——但在这个过程中，你也会失去 Tom 之前在 San Francisco 的所有信息。

对于不可变模式，情况则完全不同。不可变模式并不是存储当前系统的快照，这是可变模式做的。不可变模式是每次用户信息有变化时，就创建一个单独的记录。完成该项工作需要两个变化：第一，用一个单独的表追踪用户信息的每个字段；第二，当信息真实时，及时将每个数据的单元和一个时刻连接在一起。图2-9显示了存储 FaceSpace 信息所对应的不可变模式。



图 2-9 FaceSpace 用户信息等价的不可变模式。每个字段都会在一个单独的表中被追踪到，当数据真实的时候，每一行都会有一个时间戳（由于空间有限，性别和公司数据被忽略，但存储方式类似）

Tom 于 2012 年 4 月 4 日注册了 FaceSpace，并提供了他的个人信息。你首先了解到，这些数据的时间戳反映了记录的时间。当他后来于 2012 年 6 月 17 日搬到了 Los Angeles 时，你需要在位置表中添加一个新的记录，时间戳是他改变个人资料的时间，如图 2-10 所示。

现在有 Tom 的两个位置记录（用户 ID 为 3），并且因为数据单元与特定的时间绑定，所以它们都是真实的。Tom 的当前位置涉及对数据的简单查询：查看所有位置，并选择有最近时间戳的位置。通过在一个单独的表中保存每个字段，你只需要记录更改的信息。这需要更少的存储空间，并保证每条记录都是新的信息，而不仅仅是从最后一条记录转入。

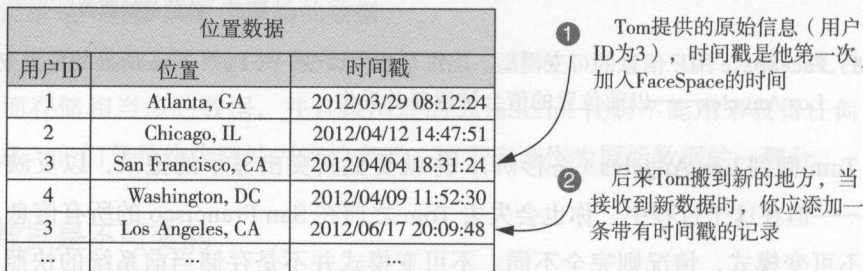


图 2-10 不可变模式不是更新现有的记录，而是使用新记录来表示更改的信息。因此不可变模式可以存储相同用户的多个记录（其他表省略，因为它们保持不变）

不可变模式的代价之一是它比可变模式使用了更多的存储。首先，用户 ID 对于每一个属性都是特定的，而不像可变模式那样一行一个。此外，存储的是所有时间段的事件，而

不是系统的当前状态。但是被称作“大数据”并不是毫无原因的。你应该充分利用大数据技术的能力来存储海量数据，以获得不变性的优势。当然，也不能夸大一个简单而较能容忍人为错误的主数据集的重要性。

2.1.3 数据是永远真实的

不变性的主要结果是，每一块数据都是永远真实的。也就是说，一块数据一旦真实，那么就必须始终真实。如果不具备这个性质，数据的不可变性就无从谈起了。你已经了解了通过打上时间戳来标记一块数据——这是一种使数据永远真实的可行方法。

这种思路和你在学校学历史是一样的。由于“1776年7月4日，美国包括13个州”这一信息总是正确的，因此“自那时以来州的数量已经增加”的事实就被捕获到额外的（也是永久的）数据中了。

一般来说，通过添加新的不变的和永远真实的数据，主数据集会持续增长。但是在一些特殊的情况下，你需要删除数据，这些情况和数据的永远真实性并不矛盾。具体情况如下：

❑ **垃圾回收**——当执行垃圾回收操作时，你应删除所有价值较低的数据单元。你可以使用垃圾回收，来实现控制主数据集增长的数据保留策略。例如，你可以决定实现一个策略，即每人每年仅保留一个位置，而不是每次用户更改位置的全部历史。

❑ **法规**——在特定条件下，政府法规可能会要求清除数据库中的数据。

在这两种情况下，删除数据不是对数据真实性的声明。相反，它是对数据价值的声明。虽然数据永远是正确的，但是你可能更愿意“忘记”一些信息，要么是因为你必须这么做，要么是因为相对于所花费的存储成本它不能实现足够的价值。

我们将使用数据的这些关键属性，继续引入另一个数据模型。

删除不可变数据？

你可能会疑惑，我们怎么才能删除不可变的数据。从表面上看，这似乎是矛盾的。一个很重要的方面是要区分对待，这里所说的“删除”是一种特殊的和少见的情况。在正常使用中，数据是不可变的，并且需要通过采取一些强制策略（如设置适当的权限），来确保其不可变。由于删除数据的情况很少发生，因此要非常谨慎地确保它能够安全删除。首先需要生成“坏的”数据被过滤掉的主数据集的另一个副本，其次运行分析工作来验证正确的数据是被过滤后的，最后再取代主数据集的旧版本——通过上述操作步骤来完成数据删除是最安全的方式。

2.2 基于事实的数据表示模型

数据是不能从其他地方衍生的信息的集合，但是在主数据集内，你可以用很多种方法来表示数据。除了传统的关系型表之外，结构化的 XML 和半结构化的 JSON 文档也是可以用于存储数据的。然而，无论如何，我们都推荐基于事实的数据模型。在基于事实的数据模型中，我们把数据分解成（毫无疑问地）名为**事实**的基本单元。

在不可变性的讨论中，你简单了解了基于事实的模型。在该模型中，主数据集会不断增加额外、不可变的、带有时间戳的数据。下面对已经讨论过的内容进行扩展，对基于事实的模型进行全面解释：首先，在 FaceSpace 示例的内容中介绍该模型并讨论其基本属性；其次，继续讨论如何，以及让事实成为可识别的原因；最后，将解释使用基于事实模型的好处，以及它为什么对主数据集是一个很好的选择。

2.2.1 事实的示例及属性

图 2-11 描述了 FaceSpace 数据中关于 Tom 的事实示例以及事实的两个核心属性——它们具有**原子性**，并带有**时间戳**。

事实具有原子性，因为它们不能再进一步被细分成有意义的组件。集合数据（如图 2-11 中 Tom 的朋友列表）可以表示为多个独立的事实。由于结果具有原子性，因此不同的事实之间没有冗余的信息。鉴于之前对数据的讨论，事实带有时间戳应该不足为奇——时间戳使得每个事实都是不可变的且永远真实的。

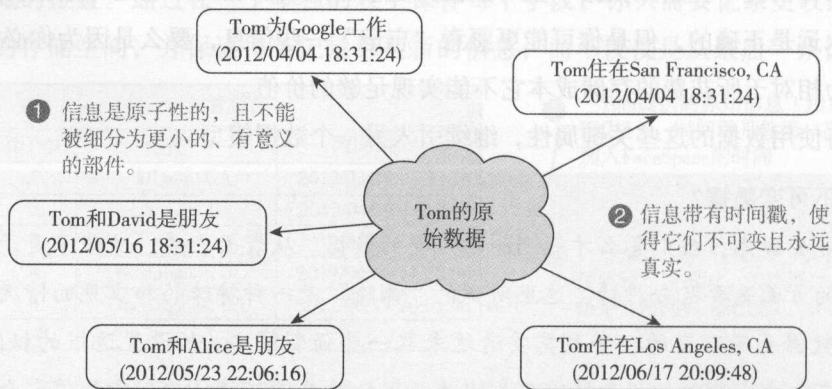


图 2-11 所有关于 Tom 的原始数据被分解为名叫**事实**的有时间戳的原子单元

对于数据集来说，这些属性使得基于事实的模型显得简单而富有表现力。我们还建议把一个额外的属性加在事实上——**可识别性**。

使事实可识别：除了原子性和时间戳之外，事实应该与一块唯一带有标识的数据相关

联。这通过例子最容易解释。

假设有存储 FaceSpace 页面浏览量的数据，你可能采用如下方法（以伪代码的形式给出）：

```
struct PageView:
    DateTime timestamp
    String url
    String ip_address
```

使用这个结构的事实数据不能唯一标识一个特定的页面浏览事件。如果多个页面浏览是在相同的时间、相同的 URL 从相同的 IP 地址传入的，那么每个页面浏览将拥有完全相同的数据记录。因此，如果你遇到两个相同的页面浏览记录，就无法区分它们指的到底是两个不同的事件，还是被意外引入数据集的重复记录。

为了区分不同的页面浏览，可以给模式添加一个区分标志——为每个页面浏览随机生成一个 64 位的数值：

```
struct PageView:
    Datetime timestamp
    String url
    String ip_address
    Long nonce
```

← 区分标志结合其他字段，唯一地
标志出一个特定的页面浏览

额外的区分标志使得彼此区分页面浏览事件成为可能，如果两个页面浏览的数据单元是相同的（所有字段，包括区分标志），那么它们指的就是完全相同的事件。

让事实成为可识别的，意味着你可以多次写同样的事实到主数据集，且不需改变主数据集的语义。在做计算时，查询可以过滤掉重复的事实。事实证明并且稍后你也将看到，可区分的事实使得实现 Lambda 架构的剩余部分更加容易。

重复数据并不像你想象得那样少

第一次看的时候，我们如此在意可识别性和重复性的目的可能不是很明显。毕竟，为了避免重复，第一反应是确保一个事件只被记录一次。但是，当处理大数据时，事情并不总是这么简单。

一旦 FaceSpace 变得更大，它将需要数百甚至数千台的 Web 服务器。建立主数据集需要将每一台服务器的数据聚合到一个中心系统——这不是一项简单的任务。有一些适合这种情况的数据收集工具，如 Facebook 的 Scribe、Apache Flume、syslog-ng 以及很多其他工具，但任何解决方案都必须是带有容错性的。

这些系统必须预先假设的一个常见的“错误”——存储数据的目标网络分区变得不可用。在这些情况下，容错系统通常会通过重试来处理失败的操作，直到它们成功。由于发

送者不知道哪些数据是失败之前收到的，因此一个标准的方法是重新发送那些没有被接收者接收的所有数据。但是如果最初尝试的一部分数据已经进入元存储，那么最终会造成数据集中的数据重复。

有一些方法可以将这些类型的操作变为事务性的，但是相当复杂且是以损失性能为代价的。在系统中确保正确性的一个重要方法是避免棘手的解决方案。通过包含可区分的事实，免去了事务性地追加数据到主数据集的需要，使得整个系统的正确性更容易保证。毕竟，如果数据模型的一个小调整可以完全避免这些挑战，为什么还要给自己施加重负呢？

快速回顾基于事实的模型的特点：

- ❑ 将原始数据存储为原子事实
- ❑ 通过使用时间戳保证事实的不变性和永远正确性
- ❑ 确保每个事实是可区分的，这样查询过程可以区分重复

接下来，我们将讨论为主数据集选择基于事实模型的优势。

2.2.2 基于事实的模型的优势

使用基于事实的模型，主数据集将是一个关于不可变的原子事实日益增长的列表。这不是关系型数据库内置支持的——如果你有关系型数据库的知识背景，你可能感到天旋地转。好消息是，通过改变数据模型范例，你可以获得许多优势。具体来说，数据有以下特点：

- ❑ 任何时刻的历史信息都是可查询的
- ❑ 容忍人为错误
- ❑ 只需要处理部分信息
- ❑ 拥有规范和规范形式的所有优点

让我们依次来看看这些优点。

1. 数据集任何时刻的历史都是可查询的

不只是像使用可变的模式那样存储系统的当前状态，你可以在数据集中查询任何时间的数据。这是具有时间戳和不可变性的事实的直接结果。通过添加带最近的时间戳的新事实，来执行“更新”和“删除”操作，而且因为没有数据被实际移除，你可以通过查询指定的时间，重建该系统的状态。

2. 数据是容忍人为错误的

容忍人为错误是通过简单地删除任何错误的事实来实现的。假设你错误地存储了 Tom

从 San Francisco 搬到 Los Angeles 的信息，如图 2-12 所示。

位置数据		
用户ID	位置	时间戳
1	Atlanta, GA	2012/03/29 08:12:24
2	Chicago, IL	2012/04/12 14:47:51
3	San Francisco, CA	2012/04/04 18:31:24
4	Washington, DC	2012/04/09 11:52:30
3	Los Angeles, CA	2012/06/17 20:09:48
...

通过简单地删除错误的信息，可以很轻松地纠正人为错误。通过使用较早的时间戳，该记录被自动重置。

图 2-12 为了纠正人为错误，可以简单地移除不正确的事实。该过程通过“暴露出”所有相关的之前的事实，来自动重置到前一个状态

通过删除 Los Angeles 的事实，Tom 的位置会自动“重置”，这是因为 San Francisco 的事实成了最新的信息。

3. 数据集能简单处理部分信息

每个记录存储一个事实，这样使得处理部分信息变得更容易——这些信息是关于无须引入 NULL 值到数据集的实体的。假如 Tom 提供了他的年龄和性别，而没有提供他的位置和职业，那么数据集将只有已知信息的事实——任何“缺失”的事实将在逻辑上等效为 NULL。一段时间后，Tom 提供的额外信息将自然而然地通过新的事实引入。

4. 数据存储层和查询处理层是独立的

基于事实的模型的另一个重要优势在一定程度上取决于 Lambda 架构本身的结构。通过在批处理层和服务层存储信息，你可以用规范化和非规范化的形式保存数据，并且同时拥有这两种形式的优点。

规范化是一个被反复使用的术语：数据规范化与之前使用的语义规范化术语是完全无关的。在这种情况下，数据规范化是指以结构化的方式存储数据，来减少冗余和促进一致性。

让我们用一个涉及关系型表格的例子来做好准备——数据规范化的内容是最常见的。关系型表格需要你在规范和非规范模式之间进行选择，基于什么对你最重要的（查询效率或数据一致性）。假设你想要存储不同兴趣的人的就业信息，图 2-13 提供了一个适合这一目的、简单非规范化的模式。

在该非规范化的模式中，同样的公司名称可能会存储多个行。这将让你能够快速确定每个公司的员工数量，但是一旦公司改了名称，你需要更新很多行。将信息存储在多个位置会增加数据不一致的风险。

就业		
行ID	名称	公司
1	Bill	Microsoft
2	Larry	BackRub
3	Sergey	BackRub
4	Steve	Apple
...

这张表格中的数据是非标准化的，因为冗余地存储了相同的信息——比如，公司名称是重复的。

你可以用这张表快速确定每个公司中员工的数量，但是如果发生变化，会有若干行需要更新——比如，BackRub修改为Google。

图 2-13 存储就业信息的一个简单非规范化的模式

作为比较，下面来看一下图 2-14 中的规范化模式。

用户		
用户ID	姓名	公司ID
1	Bill	3
2	Larry	2
3	Sergey	2
4	Steve	1
...

公司	
公司ID	姓名
1	Apple
2	BackRub
3	Microsoft
4	IBM
...	...

对于标准化的数据，每条信息只存储在一个地方并且用两个数据集之间的关系来响应查询。这简化了数据的一致性，但是表的连接操作是很昂贵的。

图 2-14 两张存储相同就业信息的规范化的表

规范化模式中的数据仅存储在一个位置。如果将 BackRub 改为 Google，那么只有 Company 表中的一行需要修改。这消除了不一致的风险，但你必须连接两张表来响应查询——这是潜在的昂贵计算。

规范和非规范模式之间的互斥选择是必要的，因为对于关系型数据库，查询是直接存储在数据层面执行的，所以你必须权衡查询效率与数据一致性的重要性，并在两种模式之间进行选择。

相比之下，查询处理和数据存储的目标在 Lambda 架构中是清楚地被分离开的，如图 2-15 所示。

在 Lambda 架构中，主数据集是完全规范化的——就像你在基于事实的模型的讨论中看到的，没有数据是冗余存储的。因为添加一个新的带有时间戳的事实“覆盖”了任何以往相关的事实，所以更新是很容易处理的。

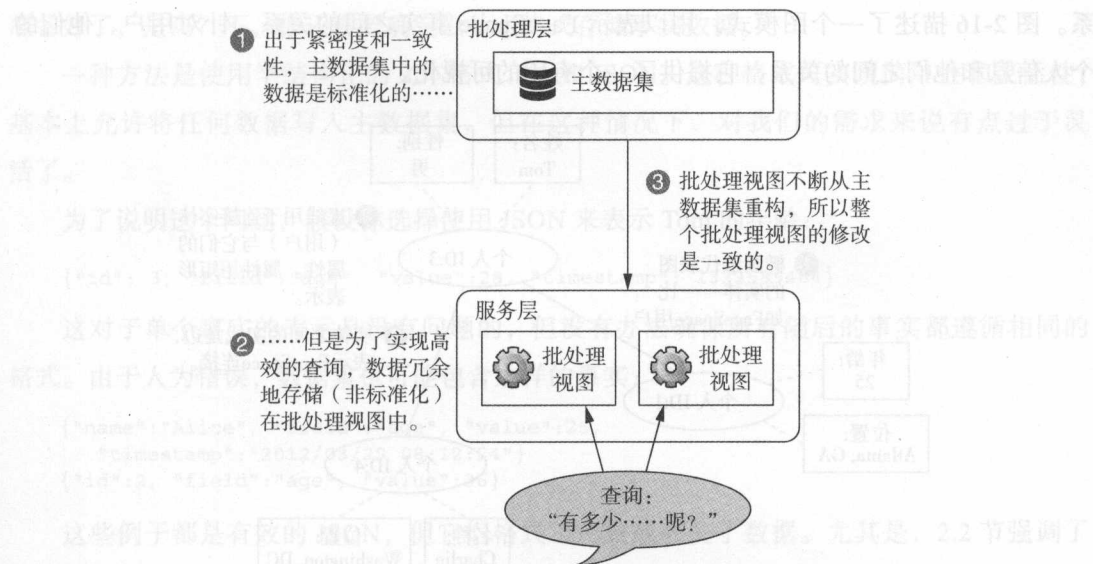


图 2-15 Lambda 架构通过在不同的层分离目标获得了规范化和非规范化的所有优点

同样，批处理视图就像主数据集中一块数据的非规范化的表，主数据集可能会被索引为许多批处理视图。关键的区别是，批处理视图被定义为主数据集上的函数，因此，没有必要更新批处理视图，因为它将不断从主数据集重建。还有一个额外的好处就是批处理视图和主数据集将永远不会失去同步。Lambda 架构以不同的方式来优化查询，拥有索引数据的性能优势，提供了完全规范化的优势。

总的来说，所有这些优势使得基于事实的模型是主数据集非常好的选择。在理论层面的讨论已经够多了，下面让我们深入实际，实现一个基于事实的数据模型的细节。

2.3 图模式

基于事实的模型中的每个事实都捕捉了一段信息。但事实本身不能表达数据背后的结构。也就是说，数据集中没有包含事实类型的描述，也没有它们之间关系的任何解释。本节将介绍图模式——图捕获使用基于事实的模型进行存储的数据集的结构。我们将讨论图模式的元素，以及使得一个模式可实施的需要。

让我们首先将 FaceSpace 的事实构建成一个图。

2.3.1 图模式的元素

2.2 节详细讨论了 FaceSpace 事实。每个事实都代表一条用户信息或两个用户之间的关

系。图 2-16 描述了一个图模式，用以表示 FaceSpace 事实之间的关系，针对用户、他们的个人信息和他们之间的关系，它提供了一个有用的可视化。

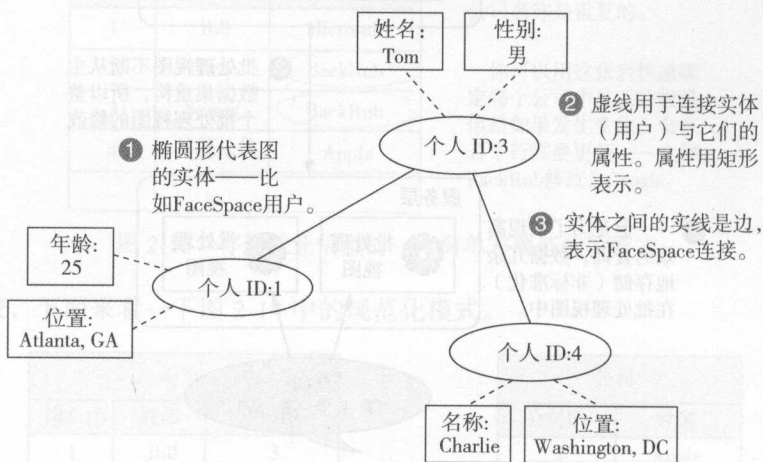


图 2-16 可视化 FaceSpace 事实之间的关系

该图强调了图模式的三个核心组件——节点、边和属性：

- **节点是系统中的实体。**在这个例子中，节点是 FaceSpace 用户，由用户 ID 表示。再如，如果 FaceSpace 允许用户将自己看作一个小组的一部分，那么该小组也将由节点表示。
- **边是节点之间的关系。**在 FaceSpace 中，边的含义是显而易见的——用户之间的边代表了 FaceSpace 朋友关系。以后你可以在用户之间添加额外的边类型，用以标识同事、家人或同学。
- **属性是关于实体的信息。**在这个例子中，年龄、性别、位置和所有其他个人信息都是属性。

节点之间的边是严格的：尽管在图 2-16 中，属性和节点看起来是连接的，但这些线不是边。它们只是用于辅助说明用户及其个人信息之间的关系。我们用实线代表边、虚线代表属性连接，来表示它们之间的不同。

图模式对数据集包含的所有数据提供了一个完整的描述。接下来将讨论确保数据集内的所有事实都严格遵循该模式的必要性。

2.3.2 可实施模式的必要性

这时，信息被存储为事实，并且用图模式描述了数据集包含的事实类型。你觉得都

准备好了，是吗？不，你还需要确定以什么格式存储事实数据。

一种方法是使用半结构化的文本格式，比如 JSON。这种格式可实现简单性和灵活性，基本上允许将任何数据写入主数据集，但在这种情况下，对我们的需求来说有点过于灵活了。

为了说明这个问题，假设你选择使用 JSON 来表示 Tom 的年龄：

```
{ "id": 3, "field": "age", "value": 28, "timestamp": 1333589484 }
```

这对于单个事实的表示是没有问题的，但没有办法确保所有随后的事实都遵循相同的格式。由于人为错误，数据集也可能包含这样的事实：

```
{ "name": "Alice", "field": "age", "value": 25,
  "timestamp": "2012/03/29 08:12:24" }
{ "id": 2, "field": "age", "value": 36 }
```

这些例子都是有效的 JSON，但它们格式不一致或丢失了数据。尤其是，2.2 节强调了每个事实有一个时间戳的重要性，但文本格式不能执行此要求。为了有效地使用数据，你必须为数据集的内容提供保障。

另一种方法是使用一个可实施的模式来严格定义事实的结构。可实施的模式预先需要做更多的工作，但是它们保证所有必需的字段都存在，并确保所有值是预期的类型。有了这些保证，开发人员将对他们期待的数据充满自信——每个事实都有一个时间戳，用户的名字永远是一个字符串，等等。关键之处在于，当创建一块数据发生错误时，可实施的模式会即刻给出错误提示，而不是稍后有人试图在不同的系统中使用数据时才发现错误。错误越早暴露为 bug，就越容易捕捉和修复。

在第 3 章中，你将看到如何使用序列化框架来实现一个可实施的模式。序列化框架提供了一种与语言无关的方式来定义模式中的节点、边和属性，然后生成代码（可能以很多不同的语言），用来序列化和反序列化模式中的对象，这样它们就可以存储到主数据集并被检索到。

此时你可能很想知道细节，不要担心——最好的学习方式是实践。在 2.4 节中，我们将在 SuperWebAnalytics.com 实体中设计基于事实的模型，并在第 3 章中使用序列化框架来实现它。

2.4 SuperWebAnalytics.com 的完整数据模型

本节旨在用 SuperWebAnalytics.com 示例将本章的所有内容联系起来。我们将从图 2-17 开始，它包含了适合我们目标的一个图模式。

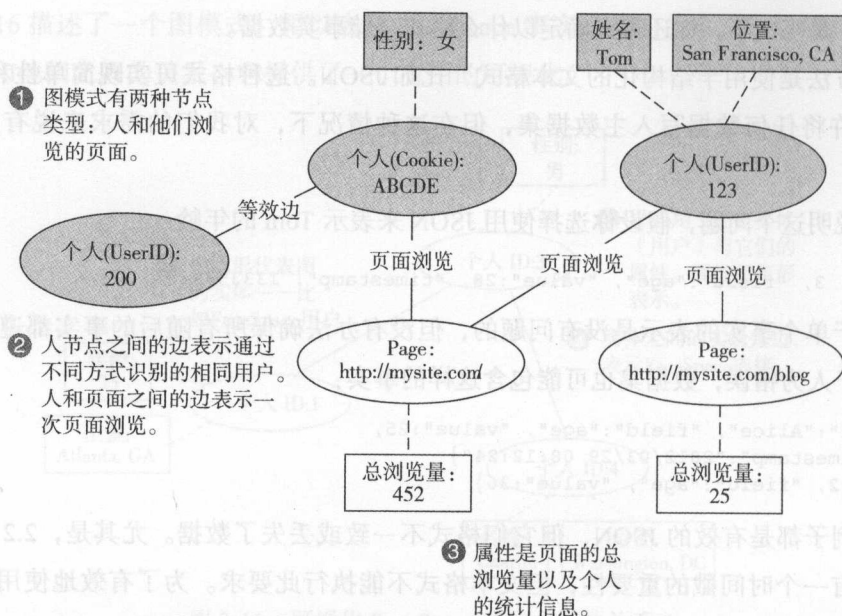


图 2-17 SuperWebAnalytics.com 的图模式，其中有两种节点类型：人和页面。人节点和它们的属性是有阴影的，用以区分这两类节点

在该模式中有两种类型的节点：人和页面。正如你所看到的，这里有两种截然不同类别的人节点，用以区分有已知标识的人和只能使用 Web 浏览器 cookie 识别的人。

该模式中的边相当简单。页面浏览的边发生在人和页面之间的每次独立访问，而当两个人节点代表了相同的一个人时，他们之间就产生了等效边。当一个最初只能通过 cookie 确认的人在稍后的时间内被完全识别时，就会发生等效边的情况。

属性也是不需要加以说明的。页面有总浏览数，人有基本的人口统计信息（姓名、性别和位置）。

基于事实的模型和图模式的优点之一是，它们可以改变不同类型的数据使其变得可用。图模式为任意不同的数据提供了一个一致的接口，所以很容易吸收新类型的信息。通过定义新的节点、边和属性类型来完成模式添加。由于事实的原子性，这些增加不会影响之前存在的事实类型。

2.5 总结

主数据集建模的方式奠定了大数据系统的基础。关于主数据集的决策确定了你可以在数据上执行什么种类的分析以及如何使用这些数据。主数据集的结构必须支持存储数据类

型的演变，因为多年来企业的数据类型可能发生大幅改变。

基于事实的模型提供了一个简单而富有表现力的数据表示，随着时间的推移，自然而然地保持了每个行为的完整历史。其“只追加”的特性使得它很容易在分布式系统中实现，并且可以很容易地实现演变数据和适应需求的改变。你不仅以更加可扩展的方式实现了关系型系统，还为系统添加了全新的功能。

当创建一个可实施的模式时，你会在写数据时获得错误——返回完整的关于数据如何以及为什么变得无效的上下文信息。图 2-1 展示了如何避免通过写入这些数据来破坏主数据表。

序列化框架是一种可实现可实施模式的简单途径。如果你曾使用过面向对象和静态类型语言，那么会很快熟悉序列化框架的使用方法。序列化框架可以生成任何你希望用来读写和验证可匹配模式的对象的代码。

然而，当试图实现全面严格的模式时，序列化框架是有局限性的。讨论完如何将序列化框架应用到 SuperWebAnalytics.com 数据模型之后，我们将讨论这些局限性及其解决方法。

3.2. Apache Thrift

Apache Thrift (<http://thrift.apache.org/>) 是一个可以用来定义静态类型结构、可实施模式、跨语言的数据访问接口。它提供了一种简单的方式来描述数据，并允许你使用不同的语言来生成客户端和服务器代码。图 2-2 展示了如何使用 Thrift 来定义一个简单的数据模型。

我们对 Apache Thrift 的使用：Thrift 最初由 Facebook 开发，后来被捐赠给 Apache 基金会。它提供了一种简单的方式来描述数据，并允许你使用不同的语言来生成客户端和服务器代码。图 2-2 展示了如何使用 Thrift 来定义一个简单的数据模型。

Thrift 的核心是结构体 (Struct) 和联合体 (Union) 的类型定义。它们是由其他数据类型组成的，如：

图 2-3 展示了如何使用 Thrift 来定义一个简单的数据模型。图 2-3 展示了如何使用 Thrift 来定义一个简单的数据模型。

Chapter 3 第3章

大数据的数据模型：示例

本章内容

- ❑ Apache Thrift
- ❑ 使用 Apache Thrift 实现图模式
- ❑ 序列化框架的局限性

在第2章中，你知道了形成一个数据模型的原则——原始数据的价值、处理语义规范化和不变性的至关重要作用。你知道了图模式可以满足所有这些属性，并了解了 SuperWebAnalytics.com 的图模式。

本章是第一个示例章节，将使用现实世界的工具演示前一章的概念。你可以只阅读本书的理论章节，学习整个 Lambda 架构，但示例章节展示了将理论转换为真正代码的细微差异。在本章中，我们将使用 Apache Thrift 实现 SuperWebAnalytics.com 的数据模型——Apache Thrift 是一个序列化框架。你会看到，即使在一个类似写模式的简单任务中，理想化的理论和实践中的实现之间也是有冲突的。

3.1 为什么使用序列化框架

许多开发人员都曾将原始数据写为无模式的格式，如 JSON。这是有吸引力的，因为它很容易入门，但这种方法很容易会导致问题。无论是由于错误还是不同开发人员之间的误解，数据损坏会不可避免地发生。根据经验，数据损坏错误是调试起来最耗时的

错误之一。

数据损坏问题很难调试，因为你很难知道损坏是如何发生的。通常只有当处理的下游出现错误时，你才会注意到这一问题——在损坏的数据已经写入很长一段时间之后。例如，由于必需字段的丢失，你可能会得到一个空指针异常。你很快就会意识到这是由一个丢失的字段引起的，但绝对不会第一时间知道数据为什么会这样。

当创建一个可实施的模式时，你会在写数据时获得错误——返回完整的关于数据如何以及为什么变得无效的上下文（类似堆栈跟踪）。此外，错误阻止了程序通过写入这些数据来损坏主数据集。

序列化框架是一种能实现可实施模式的简单途径。如果你曾使用过面向对象和静态类型语言，那么会很快熟悉序列化框架的使用方法。序列化框架可以生成任何你希望用来读、写和验证可匹配模式的对象的代码。

然而，当谈到实现全面严格的模式时，序列化框架是有局限性的。讨论完如何将序列化框架应用到 SuperWebAnalytics.com 数据模型之后，我们将讨论这些局限性及其解决方法。

3.2 Apache Thrift

Apache Thrift (<http://thrift.apache.org/>) 是一个可以用来定义静态类型化的、可实施模式的工具。它提供了接口定义语言，以通用数据类型的术语来描述模式，之后该描述可以用来自动生成多种编程语言的实现。

我们对 APACHE THRIFT 的使用：Thrift 最初由 Facebook 开发，用来构建跨语言服务。它可以用于诸多目的，但我们将缩小讨论范围，只关注它作为一个序列化框架的使用。

其他序列化框架

还有其他类似于 Apache Thrift 的工具，如 Protocol Buffers 和 Avro。记住，本书不是对每种情况的所有可能的工具提供调研，而是选用适当的工具来说明基本概念。作为一种序列化框架，Thrift 已经经过了生产环境的完全测试并得到了广泛使用。

Thrift 的核心是结构体（Struct）和联合体（Union）的类型定义。它们是由其他字段组成的，如：

□ 原始数据类型（string，integer，long 与 double）

□ 其他类型的集合（list，map 与 set）

□ 其他结构体和联合体

一般来说,用联合体来表示节点是很有用的,结构体是边的自然表示,属性则将联合体和结构体结合起来使用。这对于需要表示 SuperWebAnalytics.com 模式组件的类型定义来说是有用的。

3.2.1 节点

对于 SuperWebAnalytics.com 用户节点,通过用户 ID 或浏览器 Cookie 标识某个人,而不会同时用这两个元素标识他。这种模式对于节点是常见的,且完全匹配联合体数据类型——单个值可能有几种表示方式。

在 Thrift 中,联合体通过罗列出所有可能的表示来定义。下面的代码使用 Thrift 联合体定义 SuperWebAnalytics.com 节点:

```
union PersonID {
  1: string cookie;
  2: i64 user_id;
}

union PageID {
  1: string url;
}
```

注意: 联合体还可用于单一表现形式的节点。联合体允许模式随着数据的演变而演变——我们稍后将在本节中进一步讨论这个问题。

3.2.2 边

每条边都可以表示为包含两个节点的结构体。边结构体的名称表明了它所代表的关系,边结构体中的字段包含了参与该关系的实体。

该模式定义非常简单:

```
struct EquivEdge {
  1: required PersonID id1;
  2: required PersonID id2;
}

struct PageViewEdge {
  1: required PersonID person;
  2: required PageID page;
  3: required i64 nonce;
}
```

Thrift 结构体的字段可以表示为 required 或 optional。如果一个字段被定义为 required,

那么必须为该字段提供一个值，否则 Thrift 会在序列化或反序列化时给出一个错误。因为图模式中的每条边都必须有两个节点，所以在这个例子中它们是必需的字段。

3.2.3 属性

最后让我们来定义属性。属性包含一个节点和属性的值。因为值可以是诸多类型中的一种，所以最好使用联合体结构来表示。

首先定义页面属性的模式。页面只有一个属性，所以很简单：

```
union PagePropertyValue {
  1: i32 page_views;
}

struct PageProperty {
  1: required PageID id;
  2: required PagePropertyValue property;
}
```

接下来定义人的属性。正如你看到的，位置属性更为复杂，需要另一个结构体来定义：

```
struct Location {
  1: optional string city;
  2: optional string state;
  3: optional string country;
}

enum GenderType {
  MALE = 1,
  FEMALE = 2
}

union PersonPropertyValue {
  1: string full_name;
  2: GenderType gender;
  3: Location location;
}

struct PersonProperty {
  1: required PersonID id;
  2: required PersonPropertyValue property;
}
```

位置结构体是很有意思的，因为 city、state 和 country 字段可能被存储为单独的数据片。在这种情况下，它们是密切相关的，因此把它们都放在一个结构体作为可选字段是讲得通的。当使用位置信息时，你可能会希望得到所有字段。

3.2.4 把一切组合成数据对象

此时，边和属性被定义为不同的类型。在理想情况下，你想将所有数据存储在一起，

为访问你的信息只提供一个接口。此外，如果存储在单个数据集中，还能使得数据更容易管理。这可以通过将每个属性和边的类型封装到 DataUnit 联合体来实现——如代码清单 3-1。

代码清单 3-1 完成 SuperWebAnalytics.com 模式

```
union DataUnit {
    1: PersonProperty person_property;
    2: PageProperty page_property;
    3: EquivEdge equiv;
    4: PageViewEdge page_view;
}

struct Pedigree {
    1: required i32 true_as_of_secs;
}

struct Data {
    1: required Pedigree pedigree;
    2: required DataUnit dataunit;
}
```

每个 DataUnit 与其元数据成对保存在一个 Pedigree 结构体中。Pedigree 包含了信息的时间戳，但也可能包含调试信息或数据源。最后的 Data 结构体对应了基于行为模型中的一个行为。

3.2.5 模式演变

Thrift 的设计使得模式可以随时间而演变。这是一个至关重要的属性，因为随着业务需求的变化，你将需要添加新类型的数据，并且想尽可能轻松地这样做。

演变 Thrift 模式的关键是与每个字段相关联的数值标识符。把这些序列化形式的 ID 用于标识字段。当想要改变模式但仍要向下兼容现有的数据时，你必须遵守以下规则：

- ❑ 字段可能被重新命名。这是因为对象的序列化形式使用字段 ID 而不是名称来标识字段。
- ❑ 一个字段可能被删除，但你绝不能重用那个字段 ID。当反序列化现有数据时，Thrift 将忽略字段 ID 没有包含在模式中的所有字段。如果你重用之前删除的字段 ID，Thrift 会尝试反序列化旧数据到新的字段，这将导致无效或不正确的数据。
- ❑ 只有可选的字段可以被添加到现有的结构体。你不能添加必需的字段，因为现有的数据不会有这些字段，所以不能被反序列化。（注意：这并不适用于联合体，因为联合体没有必需和可选字段的观念）

作为一个例子，如果想要改变 SuperWebAnalytics.com 模式来存储一个人的年龄和网页之间的链接，那么应对 Thrift 定义文件做出以下改变（变化的用粗体表示）。

代码清单 3-2 扩展 SuperWebAnalytics.com

```
union PersonPropertyValue {
  1: string full_name;
  2: GenderType gender;
  3: Location location;
  4: i16 age;
}

struct LinkedEdge {
  1: required PageID source;
  2: required PageID target;
}

union DataUnit {
  1: PersonProperty person_property;
  2: PageProperty page_property;
  3: EquivEdge equiv;
  4: PageViewEdge page_view;
  5: LinkedEdge page_link;
}
```

注意：添加新的年龄属性是通过添加到相应的联合体结构完成的，并且新的边可以通过将它添加到 DataUnit 联合体来实现内嵌。

3.3 序列化框架的局限性

序列化框架只检查所有必需的字段是否存在以及是否是预期的类型。它们无法检查更丰富的属性，如“年龄应该非负”或“真实的时间戳不应该是未来的时间戳”。在系统中，与这些属性不匹配的数据会提示出现问题，而你不想让它们写入主数据集。

这看上去不像是一个限制，因为序列化框架似乎有点类似于关系型数据库模式的工作方式。事实上，你可能已经发现了使用关系型数据库模式的痛苦，并且担心更严格的模式会令人更加痛苦。但是我们要求你不要混淆使用关系型数据库模式附带的复杂性与模式本身的价值。当应用序列化框架来表示使用图模式的不可变对象时，用关系型数据库表示嵌套对象和做模式迁移会更加容易。

思考一个模式的正确方式是，将模式作为能获取一段无论是否有效的数据并返回结果的函数。ApacheThrift 的模式语言可以让你把只有字段存在和字段类型被验证过的函数表述为一个子集。理想的工具会让你实现任何可能的模式函数。

这样一个理想的工具——尤其是语言中立的——是不存在的，但你可以采取两种方法来解决序列化框架（如 Apache Thrift）的这些限制：

- ❑ 将生成的代码封装为额外的代码，用来检查你所关注的附加属性，如“年龄非负”。只要你只使用一种语言读/写数据，这种方法是适用的——如果你使用多种语言，你必须用多种语言重复该逻辑。
- ❑ 在批处理工作流的最开始检查额外的属性。这一步将把数据集分成“有效数据”和“无效数据”，如果没有任何无效的数据将会发送一个通知。这种方法使得实现工作流的其余部分更加容易，因为任何通过有效性检查的数据可以被认为拥有你所关注的更严格的属性。但是这种方法并不能阻止无效的数据写入主数据集，也不能帮助确定损坏发生时的上下文。

这两种方法都不是很理想，但如果你的结构使用多种语言读/写数据，你会发现很难做得比这更好。所以你必须决定是更愿意用多种语言保持相同的逻辑，还是失去损坏发生时的上下文。唯一完美的方法是，一个序列化框架也是一类通用的编程语言，可以将本身翻译成任何目标语言。虽然这样的工具在理论上是可能的，但在现实中是不存在的。

3.4 总结

在大多数情况下，实现 SuperWebAnalytics.com 的可实施图模式是简单的。你看到了为此使用序列化框架而出现的冲突，即无法执行你所关注的每一个属性。很少有工具能完美地符合你的要求，但重要的是要知道可能的理想工具是什么样的，这样你可以对所做出的权衡有个认识，并可以留意更好的工具（或自己做）——随着理论和示例章节的学习，这将是一个共同的主题。

第4章将介绍如何在批处理层中物理地存储主数据集，这一问题就可以被轻松、高效地处理了。

图 4-1 主数据集存储的需求和期望的批处理层

第 4 章

Chapter 4

批处理层的数据存储

本章内容

- ❑ 主数据集的存储需求
- ❑ 分布式文件系统
- ❑ 使用垂直分区提高效率

在前两章中，你了解了主数据集的数据模型以及如何将数据模型转化为图模式，也看到了使得数据保持不变性和永久性的重要性。下一步是学习如何在批处理层中真正地存储数据。图 4-1 所示即为我们在 Lambda 架构中所处的位置。

像前两章一样，这一章主要介绍主数据集。主数据集通常很大，不能存在于单个服务器上，所以你必须选择如何在多台机器上分布地存储数据。存储主数据集的方式将会影响你如何使用它，因此利用所设想的使用模式设计存储策略是至关重要的。

在本章中，你将完成如下事情：

- ❑ 确定存储主数据集的需求
- ❑ 了解分布式文件系统天生适合存储主数据集的原因
- ❑ 了解 SuperWebAnalytics.com 工程的批处理层是如何映射到分布式文件系统的

我们将通过检查 Lambda 架构中批处理层的角色如何影响存储数据的方式，来开始讨论。

4.1 主数据集的存储需求

为了确定数据存储的要求，你必须考虑如何写数据以及如何读数据。Lambda 架构中批处理层的角色影响了这两方面——我们将对此进行深入讨论，然后给出一张完整的需求列表。

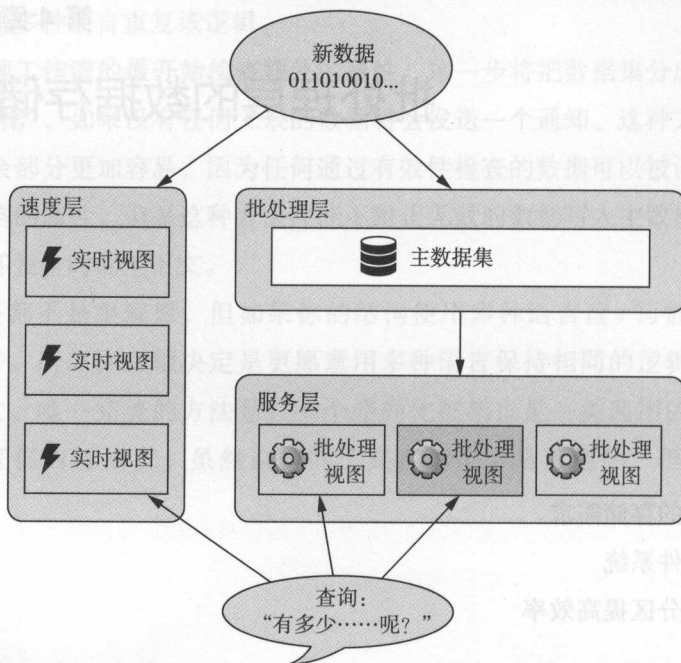


图 4-1 批处理层必须构建得很大，用来采用低维护和高效创建批处理视图的方式应对不断增长的数据集

在第 2 章中，我们强调了数据的两个关键属性：不变性和永久性。因此，每一块数据将被写入一次，且只有一次。永远不需要改变你的数据——只有写操作会添加一个新的数据单元到数据集中。因此，为了处理大量的、不断增长的数据集，存储方案必须优化。

批处理层还负责运行数据集上的函数，以生成批处理视图。这意味着需要批处理层存储系统擅长一次性读取大量数据。需要特别注意的是，对单个数据块的随机存取不是必需的。

牢记“一次写入，多次批量读取”的范式，我们就可以创建一个主数据集存储的需求代码清单，见表 4-1。

表 4-1 主数据集存储的需求代码清单

操 作	需 求	讨 论
写	高效追加新数据	唯一的写操作是添加新数据，所以将新的数据对象集添加到主数据集的操作必须是简单、高效的
	可扩展的存储	批处理层存储完整的数据集——可能是 TB 或 PB 级别的数据。因此随着数据集的增长，必须很容易扩展存储
读	支持并行处理	创建批处理视图要求计算函数基于整个主数据集，因此批处理存储必须支持并行处理，以可伸缩的方式处理大量的数据
读写	可调优存储和处理成本	存储是要花钱的。你可以选择压缩数据来降低成本，但在计算过程中解压数据会影响性能。批处理层应该能够提供灵活的机制，让你决定如何存储和压缩数据，以满足具体需求
	强制不变性	能够在主数据集上满足强制不变性是至关重要的。当然，计算机本身是可变，所以总是会有方法能够改变所存储的数据。你能做的最好的方式就是在适当的地方做检查，不允许可变的操作。这些检查应能防止缺陷或其他随机误差损坏现有数据

现在让我们看看满足这些需求的一类技术。

4.2 为批处理层选择存储方案

有了需求代码清单，现在可以考虑批处理层存储的选项。有着这样宽松的要求——甚至不需要随机访问数据——似乎你可以使用几乎任何类型的分布式数据库来存储主数据集，所以首先考虑的是使用键/值存储主数据集的可行性，这是分布式数据库最常见的类型。

4.2.1 使用键/值存储主数据集

我们还没有讨论过分布式键/值存储，但从本质上说，你可以认为它们是巨大的、持久的、分布在多台机器上的散列表。如果你将主数据集以键/值形式存储，首先必须要清楚键和值分别是什么。

值应该是什么，这是显而易见的——它是你想要存储的一段数据，但键应该是什么？在数据模型中没有天生的键，也没有必要，因为数据是按块读取的。所以你会立即遇到数据模型和键/值存储工作方式之间的阻抗失配。唯一可行的方法是生成一个通用唯一识别码 (Universally Unique Identifier, UUID) 作为键。

这只是使用键/值存储主数据集问题的开始。因为键/值存储形式需要支持对键/值的细粒度的访问，来进行随机读写，所以不能将多个键/值对压缩到一起，因此必须在存储成本和处理成本之间进行权衡。

键/值存储被用作可变的存储形式，而强制不变性对主数据集是很关键的，这是一个需要解决的问题。除非修改使用的键/值存储代码，否则通常无法禁用修改现有键/值对。

不过最大的问题是，键/值存储有很多不需要的特性：随机读、随机写和所有这些工作背后的机制。事实上，键/值存储的大多数实现是致力于这些并不需要的功能。这意味着该工具在满足需求的基础上要复杂得多，而且使用起来更有可能出现问题。此外，键/值存储对数据创建索引，且提供了不必要的服务，这将增加存储成本、降低读写数据的性能。

4.2.2 分布式文件系统

结果证明，存在一种你已经非常熟悉的技术，能够完美适用于批处理层存储——文件系统。

文件是字节序列，使用它们最有效的方法是扫描。它们按顺序存储在磁盘上（有时它们被分块存储，但从本质上说，读和写仍然是连续的）。你可以完全控制文件的每一个字节，并有充分的自由来以任何想要的格式压缩它们。与键/值存储不同，文件系统只给你所需要的（没有其他），同时也不会限制你在存储成本和处理成本之间进行权衡。最重要的是，文件系统实现了细粒度的权限系统，完美支持强制不变性。

普通文件系统的问题在于，它只存在于单台机器上，所以你只能到扩展单台机器的存储极限和处理能力。事实上，有一类称为**分布式文件系统**的技术，除了将存储分布在一个计算机集群上，它们和你熟悉的文件系统非常类似。它们通过向集群添加更多的机器来实现扩展。分布式文件系统具有容错能力，即使有一台机器出现故障（这意味着如果你失去了一台机器），你的所有文件和数据仍将可以访问到。

分布式文件系统和普通文件系统之间有一些区别。分布式文件系统的操作往往比普通文件系统更受限制。例如，在分布式文件系统中创建一个文件后，你可能无法将数据写入文件的中间部分，甚至不能修改文件。在分布式文件系统中，存储小文件是低效的，所以你要确保文件相对较大，才能充分使用分布式文件系统（具体大小与使用的工具有关，但64 MB 是很好的经验法则）。

4.3 分布式文件系统是如何工作的

很难抽象地讨论某个分布式文件系统是如何工作的，所以我们使用具体的工具来进行解释：Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS）。我们认为，HDFS 的设计充分代表了分布式文件系统的工作模式，从而说明这种工具可用于批处理层。

HDFS 和 Hadoop MapReduce 是 Hadoop 项目的两个部分：一个用于对大数据的分布式存储和分布式处理的 Java 框架。Hadoop 被部署在多台服务器上（通常被称为一个**集群**），HDFS 是分布式和可扩展的文件系统，用来管理数据是如何存储在集群的。Hadoop 这个项目在规模和深度上都很复杂，所以我们只提供高层面的描述。

在 HDFS 集群中，有两种类型的节点：一个 NameNode 和多个 DataNode。当上传一个文件到 HDFS 时，文件先被分成固定大小的块，通常在 64 MB 和 256 MB 之间；然后每一个块被复制到多个随机选取的 DataNode 上（通常是 3 个）。NameNode 负责追踪文件到块的映射（File-to-block Mapping）和每一个块的位置。设计如图 4-2 所示。

以这种方式将文件分发到多个节点，使其很容易进行并行处理。当一个程序需要访问存储在 HDFS 上的文件时，它首先与 NameNode 通信，确定哪些 DataNode 拥有要访问文件的内容。该过程如图 4-3 所示。

此外，因为每个块被复制到多个节点，所以即使个别节点离线，数据依然是可用的。当然，这种容错也有限制：如果复制因子是 3，3 个节点同时出现故障，并且你所存储的数以百万计的块中的一些恰巧存在于这 3 个节点，那么这些块将不可用。

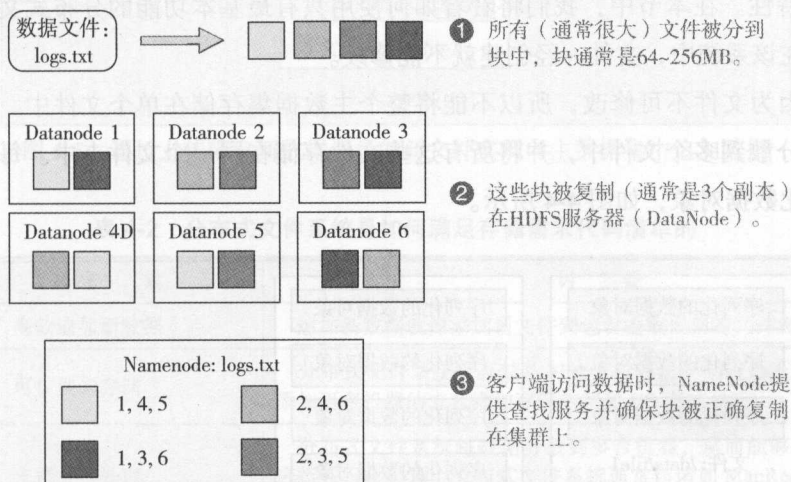


图 4-2 文件被分成块，分散存储在集群的 DataNode 上

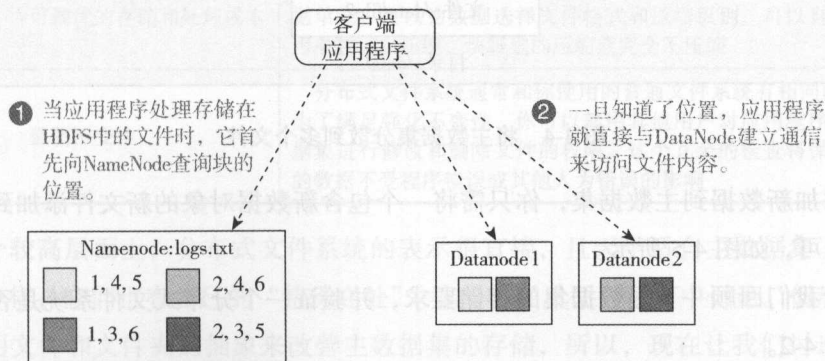


图 4-3 客户端与 NameNode 通信以确定哪些 DataNode 拥有要访问文件的块

实现一个分布式文件系统是一项艰巨的任务，但是现在你已经从用户的角度获悉完成这一任务的重点。综上所述，必须知晓的重要事项如下：

❑ 文件被分布在多台机器上以获得可伸缩性，也能够实现并行处理。

❑ 文件块被复制到多个节点，以实现容错。

下面介绍如何使用分布式文件系统存储主数据集。

4.4 使用分布式文件系统存储主数据集

不同的分布式文件系统，它们允许的各种操作之间有一些不同。一些分布式文件系统允许修改现有文件，而另一些不允许。一些分布式文件系统允许添加到现有文件，另一些则没有这个特性。在本节中，我们将看看如何使用只有最基本功能的分布式文件系统存储主数据集。在该系统中，文件一经创建就不能修改。

显然，因为文件不可修改，所以不能将整个主数据集存储在单个文件中。你能做的是将主数据集分散到多个文件中，并将所有这些文件存储在同一个文件夹中。每个文件将包含许多序列化数据对象，如图 4-4 所示。

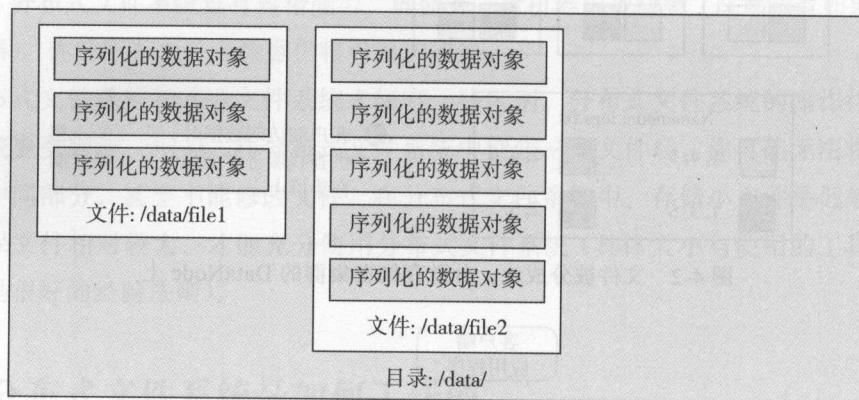


图 4-4 将主数据集分散到多个文件

若要添加新数据到主数据集，你只需将一个包含新数据对象的新文件添加到主数据集文件夹中即可，如图 4-5 所示。

现在让我们回顾一下主数据集的存储要求，并验证一个分布式文件系统是否满足这些要求，见表 4-2。

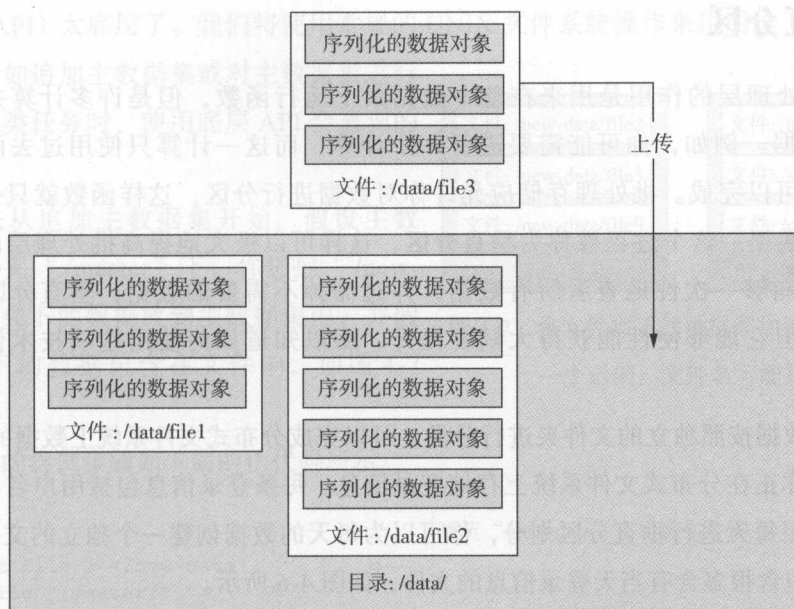


图 4-5 通过上传带有新数据记录的新文件实现向主数据集中追加数据

表 4-2 分布式文件系统是如何满足存储需求代码清单的

操 作	需 求	讨 论
写	高效追加新数据	追加新数据就像添加新文件到包含主数据集的文件夹一样简单
	可扩展的存储	分布式文件系统将存储均匀地分布在集群的机器上。你可以通过添加更多机器的方式增加存储空间和 I/O 吞吐量
读	支持并行处理	分布式文件系统将数据分散到多台机器，从而能够使用多台机器进行并行化处理。分布式文件系统通常与诸如 MapReduce 之类的计算框架集成，使得处理更容易完成（将在第 6 章讨论）
读写	可调优的存储和处理成本	就像普通文件系统，你可以完全控制用何种方式在文件中存储数据单元；可以为数据选择文件格式和压缩级别；可以自由地选择使用单独记录压缩、块级别的压缩或完全不压缩
	强制不变性	分布式文件系统通常和你使用的普通文件系统有相同的权限系统。为了满足强化不变性，你可以禁用其他用户对应用程序使用的主数据集进行修改和删除文件的权限。这个冗余的检查将保护先前存储的数据不受程序错误或其他人为错误的影响

在一个较高层面上，分布式文件系统的表示很直接，且天生适合主数据集。当然，与任何工具一样，它们也有自己的“特殊之处”，这些将在后面的章节中讨论。但事实上你仍然可以利用文件和文件夹的抽象来改善主数据集的存储，所以，现在让我们讨论使用文件夹来启用垂直分区。

4.5 垂直分区

尽管批处理层的作用是用来在整个数据集上运行函数，但是许多计算并不需要检视所有的数据。例如，你可能需要进行一个计算，而这一计算只使用过去两个星期收集的信息就可以完成。批处理存储应允许你对数据进行分区，这样函数就只会访问到与计算相关的数据。这个过程被称为**垂直分区**，这样可以极大地提高批处理层的效率。尽管批处理层能够一次性地查看所有数据，并过滤掉不需要的数据，垂直分区并不是严格必需的，但它能够使性能获得大幅度提升，所以知道如何使用这项技术仍是十分重要的。

通过将数据按照独立的文件夹进行分类，可以完成分布式文件系统上数据的垂直分区。例如，假设你正在分布式文件系统中存储登录信息。每条登录信息包括用户名、IP 地址和时间戳。按照每天进行垂直分区划分，你可以为每天的数据创建一个独立的文件夹。每个文件夹中会包含很多含有当天登录信息的文件，如图 4-6 所示。

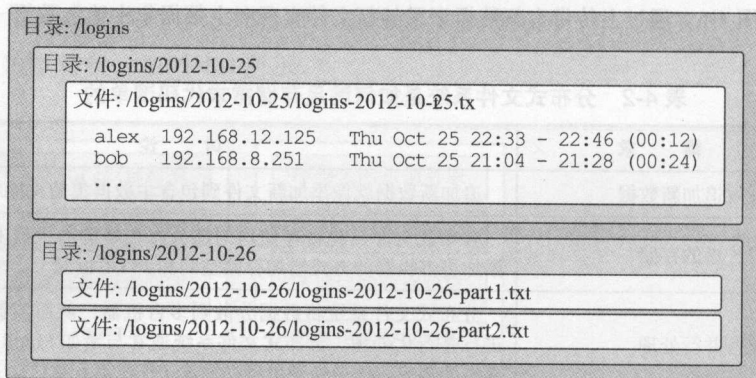


图 4-6 登录数据的垂直分区方案

通过将每天的信息分类到不同的文件夹，一个函数就可以选择只包含计算相关数据的文件夹。

现在如果你只想查看数据集的一个特定子集，那么只需查看特定文件夹下的文件并忽略其他文件夹下的文件。

4.6 分布式文件系统的底层性质

分布式文件系统提供了存储主数据集需要的存储空间和容错特性，但对于所需要执行的任务，你会发现直接使用所提供的应用程序编程接口（Application Programming

Interface, API) 太底层了。我们将使用常规的 UNIX 文件系统操作来说明这一问题, 并展示当完成诸如追加主数据集或对主数据集进行垂直分区这类任务时, 使用底层 API 会遇到的困难。

下面先从追加主数据集开始。假设主数据集在文件夹 “/master” 下, 你想把 “/new-data” 文件夹下的数据放到主数据集内, 并假设文件夹下的数据包含在文件中, 如图 4-7 所示。

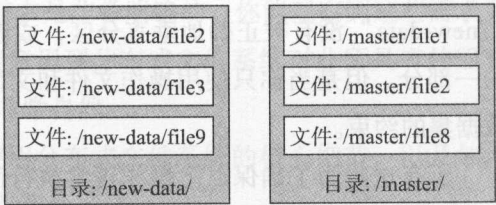
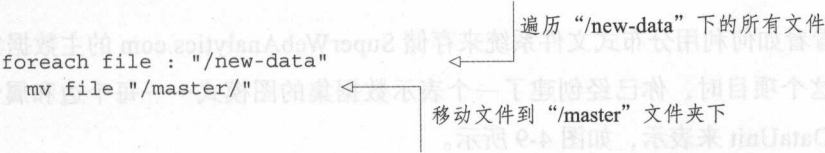


图 4-7 将文件夹下的数据添加到主数据集的一个示例。文件名可能是相同的

最直接的尝试步骤如下面的伪代码所示:



不幸的是, 这段代码有很严重的问题。如果主数据集文件夹中包含任何同名的文件, 移动操作将会失败。为了正确执行, 你必须确保将文件重命名为一个随机文件名, 以避免冲突。

还有一个问题。主数据集存储的核心需求之一是能够对存储成本和处理成本之间的取舍做出优化。当在分布式文件系统上存储一个主数据集时, 你可以选择文件格式和压缩格式来得到想要的平衡效果。如果 “/new-data” 和 “/master” 中的文件格式不同, 那么移动操作根本不会执行——这时你需要做的是将 “/new-data” 下文件中的记录复制到一个新的文件中, 并以 “/master” 下文件使用的格式进行保存。

现在使用垂直分区的主数据集完成同样的操作。假设当前的 “/new-data” 和 “/master” 如图 4-8 所示。

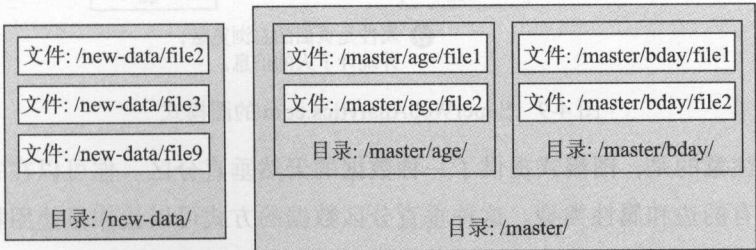


图 4-8 如果目标数据集是垂直分区的, 那么追加数据并非直接将文件添加到数据集文件夹那么简单

只是把“/new-data”中的文件放到“/master”的根目录是错误的，因为它不遵循“/master”的垂直分区规则。在这种情况下，任何追加操作都应该是不被允许的——因为“/new-data”没有被正确地垂直分区，或者对“/new-data”的垂直分区应该作为追加操作的一部分。但是当你只使用操作文件和文件夹的 API 时，它很容易出错并打破垂直分区对数据集的约束。

所有这些为了确保以上操作正常执行而需要的操作和检查，都显著表明文件和文件夹对于操纵数据集来说是一种太过底层的抽象。在接下来的章节中，你会看到一个可以自动化这些操作的库的示例。

4.7 在分布式文件系统中存储 SuperWebAnalytics.com 的主数据集

现在让我们看看如何利用分布式文件系统来存储 SuperWebAnalytics.com 的主数据集。

上次你离开这个项目时，你已经创建了一个表示数据集的图模式——每个边和属性都通过自己独立的 DataUnit 来表示，如图 4-9 所示。

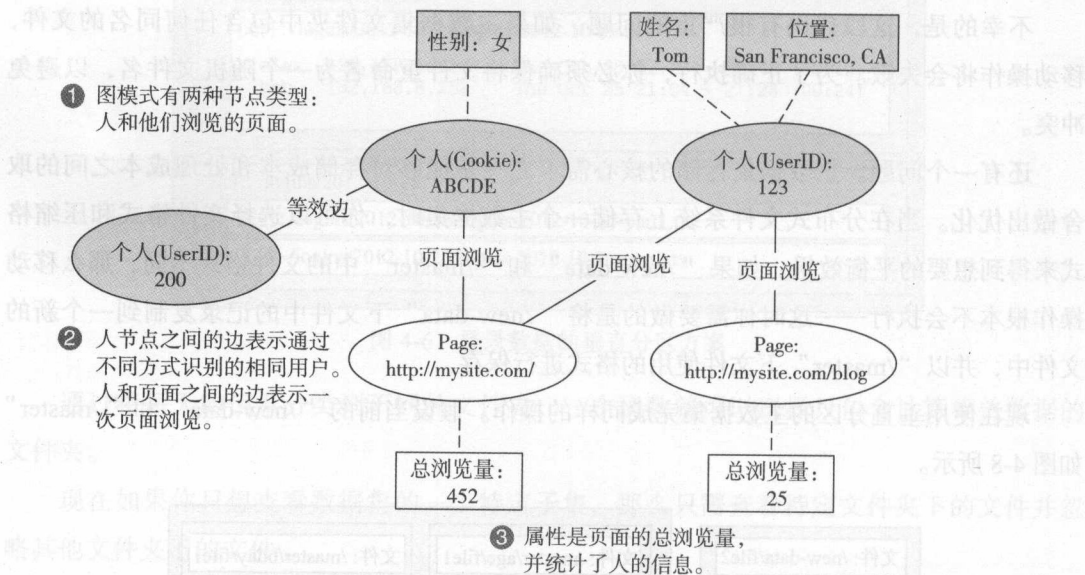


图 4-9 SuperWebAnalytics.com 的图模式

首先值得注意的是，图模式提供了一种数据的天然垂直分区。你可以在它们自己的文件夹中存储所有的边和属性类型。这种垂直分区数据的方式可以让你只使用特定的属性和边进行有效的计算。

4.8 总结

Lambda 架构中对批处理层存储数据的高级需求是非常明确的。你能观察到这些需求可以映射到一张存储方案的需求代码清单，并且能意识到分布式文件系统对此所具有的固有优势——使用和应用分布式文件系统应该会感到非常自然。

在第5章中，你将了解如何处理在实践中使用分布式文件系统的核心细节，以及如何使用更高级别的抽象来处理文件和文件夹的底层特性。

下面使用 HDFS 的 API 来处理文件和文件夹。假设你愿在一台服务器上存储所有的登录信息。以下是一些登录信息的示例。

```
$ cat /logins-2012-10-25.txt
alex 192.168.12.123 Thu Oct 25 22:33 - 22:45 (09:12)
bob 192.168.8.251 Thu Oct 25 21:04 - 21:28 (09:24)
charlie 192.168.10.52 Thu Oct 25 21:02 - 21:14 (08:12)
doug 192.168.8.18 Thu Oct 25 21:19 - 21:03 (09:23)
```

为了在 HDFS 上存储这些数据，可以为数据集创建一个目录并上传文件。使用 Hadoop 文件系统 (HDFS) 的 `hadoop fs` 命令，你可以从本地文件系统上传文件到 HDFS。例如，你可以使用以下命令将 `/logins` 目录下的所有文件上传到 HDFS 的 `/logins` 目录中：

```
$ hadoop fs -cp /logins /logins
```

你还可以使用 `hadoop fs -ls` 命令来查看 HDFS 上的文件列表。例如，你可以使用以下命令来查看 `/logins` 目录下的文件列表：

```
$ hadoop fs -ls /logins
```

输出结果如下：

```
alex 192.168.12.123 Thu Oct 25 22:33 - 22:45 (09:12)
bob 192.168.8.251 Thu Oct 25 21:04 - 21:28 (09:24)
```

```
$ hadoop fs -cat /logins/logins-2012-10-25.txt
alex 192.168.12.123 Thu Oct 25 22:33 - 22:45 (09:12)
bob 192.168.8.251 Thu Oct 25 21:04 - 21:28 (09:24)
```

正如前面所提到的，文件上传后会自动生成块，并分布在多个 DataNode 上。你可以使用以下命令来查看 HDFS 的块分布情况：

```
$ hadoop fs -ls /logins
```

输出结果如下：

```
alex 192.168.12.123 Thu Oct 25 22:33 - 22:45 (09:12)
bob 192.168.8.251 Thu Oct 25 21:04 - 21:28 (09:24)
```


Chapter 5 第5章

批处理层的数据存储：示例

4.7 在分布式文件系统上存储 SuperWebAnalytics.com 的主数据集

现在让我们看看如何利用分布式文件系统来存储 SuperWebAnalytics.com 的主数据集。上次你离开这个项目时，你已经创建了一个表示数据集的图模式——每个边和属性都通过 `DataUnit` 来表示，如图 4-9 所示。

本章内容

- ❑ 使用 Hadoop 分布式文件系统 (HDFS)
- ❑ 桶 (Pail)：用于操作数据集的更高层次的抽象

在第 4 章中，你知道了主数据集的存储需求，以及分布式文件系统是如何满足这些需求的。但是直接使用文件系统 API 对主数据集进行多种操作无疑太低级了。本章将介绍如何使用特定的分布式文件系统——HDFS，然后展示如何使用更高级的 API 使需要处理的任务实现自动化。

像所有示例章节一样，本章主要以特定的工具来展示应用第 4 章介绍的更高层次概念的细节。像往常一样，我们的目标并不是比较和对比所有可能的工具，而是强化更高层次的概念。

5.1 使用 HDFS

你已经了解了 HDFS 的基本工作原理。让我们快速回顾一下：

- ❑ 文件被拆分成块，块分散在集群的多个节点上。
- ❑ 块被复制到多个节点上，所以即使机器出现故障，数据仍然是可用的。
- ❑ NameNode 对每个文件的块以及这些块的存储位置进行追踪。

开始使用 Hadoop

准备 Hadoop 是一项艰巨的任务。Hadoop 有许多配置参数，只有针对你的硬件调整这些参数，才能获得最佳性能。为了避免被这些细节拖累，第一次接触 Hadoop 时，建议你下载一个预配置 Hadoop 环境的虚拟机。虚拟机将会加速你对 HDFS 和 MapReduce 的学习，而且有助于在你创建自己的集群时加深理解。

在撰写本书时，Hadoop 供应商 Cloudera、Hortonworks 和 MapR 都提供了公开的可用镜像。你可以获取 Hadoop 镜像，以便跟随本章及后面章节的示例进行学习。

下面使用 HDFS 的 API 来处理文件和文件夹。假设你想在一台服务器上存储所有的登录信息。以下是一些登录信息的示例：

```
$ cat logins-2012-10-25.txt
alex      192.168.12.125   Thu Oct 25 22:33 - 22:46 (00:12)
bob       192.168.8.251    Thu Oct 25 21:04 - 21:28 (00:24)
charlie   192.168.12.82      Thu Oct 25 21:02 - 23:14 (02:12)
doug      192.168.8.13       Thu Oct 25 20:30 - 21:03 (00:33)
...
```

为了在 HDFS 上存储这些数据，可以为数据集创建一个目录并上传文件：

```
$ hadoop fs -mkdir /logins
$ hadoop fs -put logins-2012-10-25.txt /logins
```

“hadoop fs” 命令是直接和 HDFS 交互的 Hadoop shell 命令。完整命令的列表在 <http://hadoop.apache.org/> 上可以找到。

文件上传后被自动分成块，并分布在 DataNode 上。

可以罗列出目录内容：

```
$ hadoop fs -ls -R /logins
-rw-r--r--  3 hdfs hadoop  175802352 2012-10-26 01:38
  /logins/logins-2012-10-25.txt
```

ls 命令基于同名的 UNIX 命令。

还可以查验文件的内容：

```
$ hadoop fs -cat /logins/logins-2012-10-25.txt
alex      192.168.12.125   Thu Oct 25 22:33 - 22:46 (00:12)
bob       192.168.8.251    Thu Oct 25 21:04 - 21:28 (00:24)
...
```

正如前面所提到的，文件上传后被自动分成块，并分布在多个 DataNode 上。可以通过以下命令确定分块情况及其块的位置：

```
$ hadoop fsck /logins/logins-2012-10-25.txt -files -blocks -locations
```

```
/logins/logins-2012-10-25.txt 175802352 bytes, 2 block(s):
OK
```

文件存储在两个块中。

```

0. blk_-1821909382043065392_1523 len=134217728
   repl=3 [10.100.0.249:50010, 10.100.1.4:50010, 10.100.0.252:50010]
1. blk_2733341693279525583_1524 len=41584624
   repl=3 [10.100.0.255:50010, 10.100.1.2:50010, 10.100.1.5:50010]

```

托管每个块的 DataNode 的 IP 地址和端口号

5.1.1 小文件问题

Hadoop 的 HDFS 和 MapReduce 紧密集成，组成了一个用于存储和处理海量数据的框架。MapReduce 将在接下来的章节中予以详细讨论；Hadoop 有一个特点——将数据存储到 HDFS 的很多小文件中时，其计算性能显著降低。相对于处理存储在几个大文件中的相同数据量的作业，处理存储在许多小文件上的 10GB 数据的 MapReduce 作业在性能上具有显著差异。

导致上述差异的原因是：一个 MapReduce 作业发起多个任务，每个任务对应输入数据集的一个块。每个任务都需要一些开销来计划和协调它的执行，并且因为每个小文件需要一个单独的任务，开销成本大量重复。MapReduce 的这个属性意味着，如果数据集中的小文件变得越来越多，你可能想要将数据合并到一起，那么你可以通过使用 HDFS API 编写代码，或使用自定义的 MapReduce 作业来实现，但是这两种方法都需要大量的工作以及你对 Hadoop 内部机制的充分理解。

5.1.2 转向更高层次的抽象

本书着重强调的一点是，解决方案不仅要可扩展、容错和性能良好，还要优雅。优雅的解决方案指的是它能够以简洁的方式来表示你所关注的计算。

当谈到操作主数据集时，你在第 4 章中看到了下面两个重要的操作：

- ❑ 对数据集进行追加
- ❑ 对数据集进行垂直分区，且不允许违背已存在的分区规则

除此之外，我们将添加一个针对 HDFS 的需求：有效地将小文件合并成较大的文件。

如第 4 章所述，直接用文件和文件夹来完成这些任务是烦琐且容易出错的。因此我们将介绍一个能够以优雅的方式完成这些任务的库。

与使用 HDFS API 的代码相比，考虑使用下面 Pail 库的代码清单。

代码清单 5-1 HDFS 维护任务的抽象

```

import java.io.IOException;
import backtype.hadoop.pail.Pail;

public class PailMove {

```

Pail 是 HDFS
文件夹的封装。

```
public static void mergeData(String masterDir, String updateDir)
    throws IOException
{
    Pail target = new Pail(masterDir);
    Pail source = new Pail(updateDir);
    target.absorb(source);
    target consolidate();
}
```

使用 Pail 库，追加是单
行代码的操作。

在 Pail 中，小文件可以通过一个函数调用
进行合并。

借助 Pail，可以用一行代码实现追加文件夹，用另一行代码合并小文件。当进行追加操作时，如果目标文件夹中的数据是不同的文件格式，Pail 会自动强制将新数据转换成正确的文件格式；如果目标文件夹有不同的垂直分区方案，Pail 将抛出一个异常。最重要的是，像 Pail 这样更高层次的抽象允许你直接使用数据，而不是使用像文件和目录这样的低级容器。

快速回顾

在学习更多关于 Pail 的知识之前，现在是你进行回顾并拓宽视野的好时机。回想一下，主数据集在 Lambda 架构中是基础的来源，并且批处理层必须成功地处理一个大的、不断增长的数据集。此外，必须有一个简单高效的方法，将数据转换成批处理视图，以响应实际查询。

这一章比之前的章节更具技术性，但要记住这一切是如何集成在 Lambda 架构中的。

5.2 使用 Pail 在批处理层存储数据

Pail 是 dfs-datastores 库（[http:// github.com/nathanmarz/dfs-datastores](http://github.com/nathanmarz/dfs-datastores)）中对文件和文件夹的一层弱抽象。这种抽象使它更容易管理批处理记录的集合。顾名思义，Pail 使用桶和文件夹保存关于数据集的元数据。通过使用元数据，Pail 允许你安全地操作批处理层，而不用担心违反其完整性。Pail 的目标只是让你关注的操作——包括对数据集进行追加、垂直分区和合并——能够安全、简单和高效地执行。

从内部原理来说，Pail 就是一个使用标准 Hadoop API 的 Java 库。它处理底层文件系统的交互，提供一个隔离 Hadoop 内部复杂性的 API。其目的是让你关注数据本身，而不是如何存储与维护数据。

为什么关注 Pail？

Pail 以及这本书涉及的许多其他包，都是由 Nathan 在开发 Lambda 架构时编写的。我们介绍这些技术不是为了推广它们，而是讨论它们的起源和所解决的问题。因为 Pail

是由 Nathan 开发的，所以到目前为止，它完美地匹配了主数据集的需求，且这些需求是从查询所有数据的函数的基本原理中自然而然出现的。你可以随意使用其他库或自己开发——重点是展示一种具体的方法来使用现有工具构建大数据系统的概念。

你已经知道了 HDFS 的特点，这一特点使它成为在批处理层存储主数据集的一个可行选择。当学习 Pail 时，你要记住它在简化数据操作的同时是如何保持 HDFS 优势的。在讨论了 Pail 的基本操作之后，我们将总结该库的总体价值。

现在让我们来看看创建和写数据到桶时，Pail 是如何工作的。

5.2.1 Pail 基本操作

理解 Pail 是如何工作的最佳方法是在计算机上跟踪并运行其提供的代码。要做到这一点，你需要从 GitHub 上下载源代码并构建 dfs-datastores 库。如果没有 Hadoop 集群或可用的虚拟机，那么在示例中本地文件系统将被视为 HDFS，然后你可以通过查看文件系统上的相关目录来观察这些命令的结果。

下面从创建一个新 Pail 与存储一些数据开始：

```

为 Pail 中的新文件提供一个输出流。
public static void simpleIO() throws IOException {
    Pail pail = Pail.create("/tmp/mypail");
    TypedRecordOutputStream os = pail.openWrite();
    os.writeObject(new byte[] {1, 2, 3});
    os.writeObject(new byte[] {1, 2, 3, 4});
    os.writeObject(new byte[] {1, 2, 3, 4, 5});
    os.close();
}
关闭当前文件。

```

在指定目录上创建一个默认 Pail 对象。

没有元数据的 Pail 仅限于存储字节数组。

当检查文件系统时，你将看到创建了文件夹“/tmp/mypail”且其中包含两个文件：

```

root:/ $ ls /tmp/mypail
f2fa3af0-5592-43e0-a29c-fb6b056af8a0.pailfile
pail.meta

```

存储在 pailfile 中的记录。

元数据描述了 Pail 的内容和结构。

pailfile 中包含刚刚存储的记录。该文件是自动创建的，因此你创建的所有记录将会立即呈现——也就是说，一个从 Pail 中读取记录的应用程序，直到文件关闭后才能看到文件。此外，pailfile 使用全局唯一名称（所以在你的文件系统上它需要被赋予不同的名称）。这些唯一的名称允许多个源同时无冲突地写入相同的 Pail 中。

目录中的另一个文件包含了 Pail 的元数据。该元数据描述了数据的类型以及数据是如何存储在 Pail 内的。当构建 Pail 时，示例没有指定任何元数据，所以该文件包含的是默认

设置：使用 Pail 进行批处理存储

```
root:/ $ cat /tmp/mypail/pail.meta
```

```
---
```

```
format: SequenceFile
```

```
args: {}
```

Pail 中文件的格式；在 Hadoop SequenceFile 中，默认的 Pail 使用键/值对存储数据。

参数描述了 Pail 的内容；一个空的 map 指向 Pail 将数据视为未压缩的字节数组。

在本章的后面，你将看到另一个包含更多元数据的 `pail.meta` 文件，但它的总体结构将保持不变。接下来介绍如何将在 Pail 中存储真正的对象，而不仅仅是二进制记录。

5.2.2 序列化对象到 Pail 中

为了在 Pail 中存储对象，你必须为 Pail 提供用于将对象序列化到二进制数据和反序列化二进制数据到对象的指令。下面通过服务器登录的示例，来演示这是如何实现的。

在下面的代码清单中，有一个简化的类，用以表示登录。

代码清单 5-2 一个用于登录的、简单的、没有修饰的类

```
public class Login {
    public String userName;
    public long loginUnixTime;

    public Login(String _user, long _login) {
        userName = _user;
        loginUnixTime = _login;
    }
}
```

为了将 Login 对象存储在 Pail 中，需要创建一个实现了 `PailStructure` 接口的类。下面的代码清单定义了 `LoginPailStructure` 类，该类描述了应该如何执行序列化。

代码清单 5-3 实现 PailStructure 接口

```
public class LoginPailStructure implements PailStructure<Login>{

    public Class getType() {
        return Login.class;
    }

    public byte[] serialize(Login login) {
        ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
        DataOutputStream dataOut = new DataOutputStream(byteOut);
        byte[] userBytes = login.userName.getBytes();
        try {
            dataOut.writeInt(userBytes.length);
            dataOut.write(userBytes);
        } catch (IOException e) {
            // ...
        }
    }
}
```

这种结构的 Pail 只存储 Login 对象。

当 Login 对象存储在 pailfile 中时，必须进行序列化。

getTarget 方法用于定义垂直分区模式, 但在这个示例中没有用到。

```

        dataOut.writeLong(login.loginUnixTime);
        dataOut.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    return byteOut.toByteArray();
}

public Login deserialize(byte[] serialized) {
    DataInputStream dataIn =
        new DataInputStream(new ByteArrayInputStream(serialized));
    try {
        byte[] userBytes = new byte[dataIn.readInt()];
        dataIn.read(userBytes);
        return new Login(new String(userBytes), dataIn.readLong());
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

public List<String> getTarget(Login object) {
    return Collections.EMPTY_LIST;
}

public boolean isValidTarget(String... dirs) {
    return true;
}
}

```

当从 pailfile 中读取 Login 对象时, Login 对象会重新创建。

isValidTarget 方法确定给定的路径是否匹配垂直分区模式, 但在这个例子中也没有用到。

将该 LoginPailStructure 类传递给 Pail 的 create 方法, 由此产生的 Pail 将使用这些序列化指令。接着可以将 Login 对象直接给 Pail, Pail 将自动处理序列化。

```

public static void writeLogins() throws IOException {
    Pail<Login> loginPail = Pail.create("/tmp/logins",
        new LoginPailStructure());
    TypedRecordOutputStream out = loginPail.openWrite();
    out.writeObject(new Login("alex", 1352679231));
    out.writeObject(new Login("bob", 1352674216));
    out.close();
}

```

使用 PailStructure 创建一个 Pail。

同样地, 当你读数据时, Pail 将反序列化记录。以下代码展示如何遍历刚写入的所有对象:

Pail 为它的对象类型提供可迭代接口。

```

public static void readLogins() throws IOException {
    Pail<Login> loginPail = new Pail<Login>("/tmp/logins");
    for (Login l : loginPail) {
        System.out.println(l.userName + " " + l.loginUnixTime);
    }
}

```

一旦数据存储在 Pail 中, 就可以使用 Pail 的内建函数对其进行安全地操作。

5.2.3 使用 Pail 进行批处理操作

Pail 对一些常用操作有内建支持。在这些操作中，你将看到使用 Pail 管理而不是手动管理记录所带来的好处。这些操作都使用 MapReduce 实现，所以不管 Pail 中的数据量是 GB 级还是 TB 级，都可以进行扩展。后面的章节将更多地讨论 MapReduce，但关键是这些操作都能够自动并行化且跨越由多个工作机器组成的集群执行。

前面的章节讨论了追加与合并操作的重要性。正如你所期望的那样，Pail 支持这两种操作——追加操作尤其巧妙。它会检查 Pail，以确认将 Pail 追加到一起时是否有效。例如，它不允许你追加包含字符串的 Pail 到包含整数的 Pail 中。如果 Pail 存储相同类型但不同文件格式的记录，它会强制进行数据转换，来匹配目标 Pail 的格式。这意味着你在存储成本和处理性能之间做出的权衡将被 Pail 强制执行。

在默认情况下，合并操作通过合并小文件来创建尽量接近 128 MB 的新文件——HDFS 标准块的大小。该操作也可以通过 MapReduce 并行执行。

来看看登录的示例，假设在一个单独的 Pail 中有额外的登录数据，并且希望将这些数据合并到原来的 Pail 中。下面的代码演示了追加与合并操作：

```
public static void appendData() throws IOException {
    Pail<Login> loginPail = new Pail<Login>("/tmp/logins");
    Pail<Login> updatePail = new Pail<Login>("/tmp/updates");
    loginPail.absorb(updatePail);
    loginPail.consolidate();
}
```

使用 Pail 进行批处理操作的主要优点是这些内建函数让你专注于“想对数据做什么”，而不用担心“如何正确地操作文件”。

5.2.4 使用 Pail 进行垂直分区

前面提到过，在 HDFS 中，可以通过使用多个文件夹对数据进行垂直分区。想象一下尝试手动管理垂直分区的场景：总是很容易忘记两个数据集的分区不同，然后错误地对它们进行追加。同样地，很容易在合并数据时不小心违反分区的结构。值得庆幸的是，Pail 能够巧妙地强制保证其结构，并规避这样的错误。

为了给 Pail 创建一个分区的目录结构，必须实现 PailStructure 接口的另外两个方法：

- ❑ **getTarget**——给定一条记录，使用 **getTarget** 方法确定该记录应该存储的目录结构，并将路径作为字符串列表返回。
- ❑ **isValidTarget**——给定一个字符串数组，使用 **isValidTarget** 方法构建一个目录路径，

并确定它是否符合垂直分区的模式。

Pail 使用上述方法来保持其结构，并自动将记录映射到其正确的子目录。

下面的代码演示了如何对 Login 对象分区，以便通过登录日期对记录进行分组。

代码清单 5-4 Login 记录的一种垂直分区模式

```
public class PartitionedLoginPailStructure extends LoginPailStructure {
    SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");

    public List<String> getTarget(Login object) {
        ArrayList<String> directoryPath = new ArrayList<String>();
        Date date = new Date(object.loginUnixTime * 1000L);
        directoryPath.add(formatter.format(date));
        return directoryPath;
    }

    public boolean isValidTarget(String... strings) {
        if(strings.length != 1) return false;
        try {
            return (formatter.parse(strings[0]) != null);
        } catch(ParseException e) {
            return false;
        }
    }
}
```

Login 被垂直分区到对应登录日期的文件夹中。

Login 对象的时间戳被转换为一种可读格式。

isValidTarget 方法验证该目录结构只具有一层深度，且文件夹名称是日期。

有了这个新的 Pail 结构，无论何时写入新的 Login 对象，Pail 均能确定其存储的正确子文件夹：

```
public static void partitionData() throws IOException {
    Pail<Login> pail = Pail.create("/tmp/partitioned_logins",
        new PartitionedLoginPailStructure());
    TypedRecordOutputStream os = pail.openWrite();
    os.writeObject(new Login("chris", 1352702020));
    os.writeObject(new Login("david", 1352788472));
    os.close();
}
```

1352702020 是 2012-11-11, 22:33:40 PST 的时间戳。

1352788472 是 2012-11-12, 22:34:32 PST 的时间戳。

检查新的 Pail 目录，以确认数据是否正确分区：

```
root:/ $ ls -R /tmp/partitioned_logins
2012-11-11 2012-11-12 pail.meta

/tmp/partitioned_logins/2012-11-11:
d8c0822b-6caf-4516-9c74-24bf805d565c.pailfile

/tmp/partitioned_logins/2012-11-12:
d8c0822b-6caf-4516-9c74-24bf805d565c.pailfile
```

Pail 中针对不同登录日期创建了不同的文件夹。

5.2.5 Pail 文件格式与压缩

Pail 将数据存储在其目录结构内的多个文件中。可以通过指定 Pail 应该使用的文件格式，来控制 Pail 如何将记录存储到这些文件中。这个特性允许你在 Pail 使用的存储空间总量和从 Pail 中读取记录的性能之间做出取舍。正如本章前面所讨论的，这个基础控制需要你进行上下调节来匹配应用程序需求。

可以实现自定义文件格式，但在默认情况下，Pail 使用 Hadoop 的 SequenceFile（序列化文件）。这种格式广泛应用，它允许一个单独的文件通过 MapReduce 并行处理，并且原生地支持文件中记录的压缩。

下面展示如何创建一个使用 SequenceFile 格式并启用 gzip 块压缩的 Pail：

```

Pail 的内容将进行
gzip 压缩。
public static void createCompressedPail() throws IOException {
    Map<String, Object> options = new HashMap<String, Object>();
    options.put(SequenceFileFormat.CODEC_ARG,
                SequenceFileFormat.CODEC_ARG_GZIP);
    options.put(SequenceFileFormat.TYPE_ARG,
                SequenceFileFormat.TYPE_ARG_BLOCK);
    LoginPailStructure struct = new LoginPailStructure();
    Pail compressed = Pail.create("/tmp/compressed",
                                  new PailSpec("SequenceFile", options, struct));
}
记录块将被压缩
在一起（与
单独行压缩
进行比较）。
创建一个新 Pail 以目标格式存储
Login 选项。

```

然后可以观察 Pail 元数据中的这些属性：

```

root:/ $ cat /tmp/compressed/pail.meta
---
format: SequenceFile
structure: manning.LoginPailStructure
args:
  compressionCodec: gzip
  compressionType: block

```

← LoginPailStructure 的
完整类名。

← pailfile 的压缩选项。

每当记录被添加到这个 Pail 中时，它们会被自动压缩。在读取和写入记录时，Pail 所使用的空间，明显更少，但所需要花费的 CPU 代价更高。

5.2.6 Pail 优点的总结

前面已经探讨了 Pail 的内部原理，重点是要理解它在原生 HDFS 之上所提供的优点。

表 5-1 针对表 4-1 中的需求代码清单总结了 Pail 存储主数据集的优点。

表 5-1 Pail 存储主数据集的优点

操 作	标 准	讨 论
写操作	高效追加新数据	Pail 对于追加数据有一个最优的接口，能避免执行无效的操作——原生 HDFS API 并没有提供这个特性
	可扩展的存储	NameNode 在内存中维护整个 HDFS 的命名空间，如果文件系统包含大量的小文件，NameNode 的负担会加重。Pail 的合并操作减少了总的 HDFS 块数，减轻了对 NameNode 的需求
读操作	支持并行处理	MapReduce 作业中任务的数量是由数据集中块的数量所决定的。合并 Pail 的内容降低了所需任务的数量，提高了处理数据的效率
	能够对数据垂直分区	输出到 Pail 的写操作是自动分区的，每条记录存储在它适合的目录中。该目录结构是所有 Pail 操作都要严格执行的
读写操作	可调优的存储/处理成本	Pail 内建支持将数据转换为 PailStructure 指定的格式。当对 Pail 执行操作时，这种强制转换是自动发生的
	强制不变性	因为 Pail 只是一层对文件和文件夹的简单封装，所以您可以通过设置适当的权限来保证强制不变性，就像直接使用 HDFS 一样

该表总结了 Pail 的优点。Pail 对批处理层中的数据交互进行了有用且强大的抽象，同时使你不需要关注底层文件系统的细节。

5.3 存储 SuperWebAnalytics.com 的主数据集

在第 4 章中，你知道了存储 SuperWebAnalytics.com 数据的高级概念是多么简单——使用分布式文件系统，通过在不同的子文件夹中存储不同的属性与边实现垂直分区。现在让我们使用你已经学过的工具，使之成为现实。

回想一下为 SuperWebAnalytics.com 开发的 Thrift 模式。该模式的摘要如下：

```

struct Data {
    1: required Pedigree pedigree;
    2: required DataUnit dataunit;
}

union DataUnit {
    1: PersonProperty person_property;
    2: PageProperty page_property;
    3: EquivEdge equiv;
    4: PageViewEdge page_view;
}

union PersonPropertyValue {
    1: string full_name;
    2: GenderType gender;
    3: Location location;
}

```

数据集中的所有数据都表示为一个时间戳和一个数据基本单元。

基础数据单元描述数据集的边和属性。

属性的值可以是多种类型。

图 5-1 展示了如何将该模式映射到文件夹。

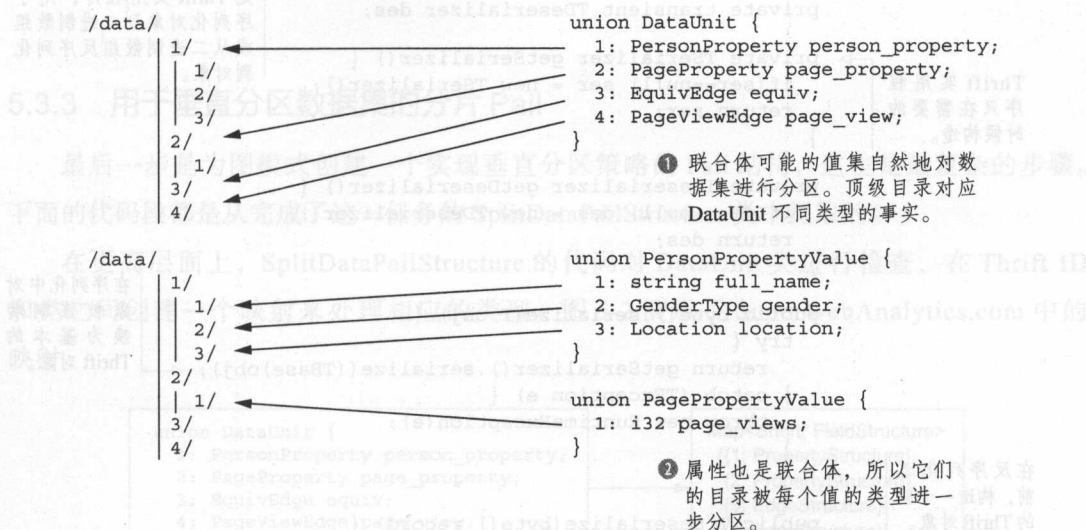


图 5-1 图模式中的联合体对数据集提供了一个自然的垂直分区模式

为了使用 HDFS 和 Pail 存储 SuperWebAnalytics.com，必须定义一个结构化的 Pail 来存储 Data 对象，同时能够维持垂直分区模式。这段代码有点复杂，所以我们分步来呈现它：

1) 首先，将创建一个用于存储 Thrift 对象的抽象 Pail 结构。Thrift 序列化独立于存储数据的类型，通过分离这一逻辑使得代码更加简洁。

2) 其次，将从上边定义好的抽象类中获得一个 Pail 结构来存储 SuperWebAnalytics.com 的 Data 对象。

3) 最后，将进一步定义一个子类，实现所需要的垂直分区模式。

在本节中，不要担心代码的细节。这段代码适用于任何图模式，甚至随着模式的发展它仍适用。

5.3.1 Thrift 对象的结构化 Pail

为 Thrift 对象创建一个 Pail 结构非常容易，因为 Thrift 帮你做了大量的工作。下面的代码清单展示了如何使用 Thrift 实体来序列化和反序列化数据。

代码清单 5-5 序列化 Thrift 对象的通用抽象 Pail 结构

```
Java 泛型允许 Pail 结构适用于任何 Thrift 对象。
public abstract class ThriftPailStructure<T extends Comparable>
    implements PailStructure<T>
{
```



```

private transient TSerializer ser;
private transient TDeserializer des;

private TSerializer getSerializer() {
    if (ser == null) ser = new TSerializer();
    return ser;
}

private TDeserializer getDeserializer() {
    if (des == null) des = new TDeserializer();
    return des;
}

public byte[] serialize(T obj) {
    try {
        return getSerializer().serialize((TBase)obj);
    } catch (TException e) {
        throw new RuntimeException(e);
    }
}

public T deserialize(byte[] record) {
    T ret = createThriftObject();
    try {
        getDeserializer().deserialize((TBase)ret, record);
    } catch (TException e) {
        throw new RuntimeException(e);
    }
    return ret;
}

protected abstract T createThriftObject();

```

Thrift 实用程序只在需要的时候构造。

TSerializer 和 TDeserializer 是 Thrift 实用程序，用于序列化对象到二进制数组或从二进制数组反序列化到对象。

在序列化中对象被强制转换为基本的 Thrift 对象。

在反序列化之前，构造一个新的 Thrift 对象。

Thrift 对象的构造方法必须在子类中实现。

5.3.2 SuperWebAnalytics.com 的基础 Pail

接下来，可以通过创建 ThriftPailStructure 的具体子类，来定义一个基本的类来存储 SuperWebAnalytics.com 的 Data 对象，如代码清单 5-6 所示。

代码清单 5-6 Data 对象的具体实现

```

public class DataPailStructure extends ThriftPailStructure<Data> {
    public Class getType() {
        return Data.class;
    }

    protected Data createThriftObject() {
        return new Data();
    }

    public List<String> getTarget(Data object) {
        return Collections.EMPTY_LIST;
    }
}

```

指定存储 Data 对象的 Pail。

需要通过 ThriftPailStructure 类创建一个对象进行反序列化。

该 Pail 结构没有使用垂直分区。

```

public boolean isValidTarget(String... dirs) {
    return true;
}

```

5.3.3 用于垂直分区数据集的分片 Pail

最后一步是为图模式创建一个实现垂直分区策略的 Pail 结构, 这也是最复杂的步骤。下面的代码段都是从完成了这一任务的 SplitDataPailStructure 类中提取的。

在更高层面上, SplitDataPailStructure 的代码对 DataUnit 类进行检查, 在 Thrift ID 和类之间创建一个映射来处理相应的类型。图 5-2 演示了 SuperWebAnalytics.com 中的映射。

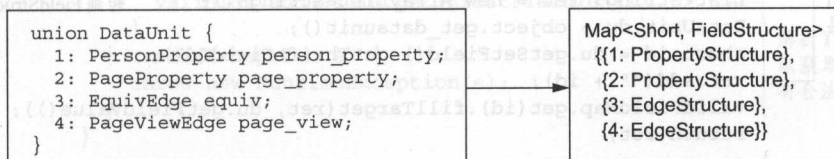


图 5-2 SuperWebAnalytics.com 的 DataUnit 类的 SplitDataPailStructure 字段映射

代码清单 5-7 包含生成字段映射的代码。它适用于任何图模式, 不仅是针对这个例子。

代码清单 5-7 为图模式生成字段映射的代码

```

public class SplitDataPailStructure extends DataPailStructure {

    Thrift 代
    码用来检
    查和遍历
    DataUnit
    对象。
    如果一个
    类名不以
    "Property"
    结尾, 那么它肯定
    是一条边。

    public static HashMap<Short, FieldStructure> validFieldMap =
        new HashMap<Short, FieldStructure>();

    static {
        for(DataUnit.Fields k: DataUnit.metaDataMap.keySet()) {
            FieldValueMetaData md = DataUnit.metaDataMap.get(k).valueMetaData;
            FieldStructure fieldStruct;
            if(md instanceof StructMetaData &&
               ((StructMetaData) md).structClass
                   .getName().endsWith("Property"))
            {
                fieldStruct = new PropertyStructure(
                    ((StructMetaData) md).structClass);
            } else {
                fieldStruct = new EdgeStructure();
            }
            validFieldMap.put(k.getThriftFieldId(), fieldStruct);
        }
    }

    // 类的剩余部分省略
}

```

FieldStructure 是边和属性的接口。

属性是通过被检查对象的类名来鉴别的。

正如代码注释所提及的，FieldStructure 是 PropertyStructure 和 EdgeStructure 的共享接口。该接口的定义如下：

```
protected static interface FieldStructure {
    public boolean isValidTarget(String[] dirs);
    public void fillTarget(List<String> ret, Object val);
}
```

稍后我们会提供 EdgeStructure 类和 PropertyStructure 类的详细信息，现在只关注该接口是如何用来完成表的垂直分区的：

// 来自 SplitDataPailStructure 中的方法

```
通过检查 DataUnit 确定高层次目录。
    public List<String> getTarget(Data object) {
        List<String> ret = new ArrayList<String>();
        DataUnit du = object.get_dataunit();
        short id = du.getSetField().getThriftFieldId();
        ret.add("" + id);
        validFieldMap.get(id).fillTarget(ret, du.getFieldValue());
        return ret;
    }
    // 任何进一步的分区都转到 FieldStructure 中。

有效性检查首先验证 DataUnit 字段 ID 是否在字段映射中。
    public boolean isValidTarget(String[] dirs) {
        if(dirs.length==0) return false;
        try {
            short id = Short.parseShort(dirs[0]);
            FieldStructure s = validFieldMap.get(id);
            if(s==null)
                return false;
            else
                return s.isValidTarget(dirs);
        } catch (NumberFormatException e) {
            return false;
        }
    }
    // 任何额外的检查都转到 FieldStructure 中。
```

SplitDataPailStructure 负责垂直分区的高层次目录，并将任何额外的子目录交给 FieldStructure 类负责。因此，一旦定义了 EdgeStructure 和 PropertyStructure 类，工作就完成了。边是结构体，因此不能再分区。这使得 EdgeStructure 类非常简单：

```
protected static class EdgeStructure implements FieldStructure {
    public boolean isValidTarget(String[] dirs) { return true; }
    public void fillTarget(List<String> ret, Object val) { }
}
```

但是属性是联合体，如 DataUnit 类。代码同样通过检查为给定的属性类创建一组有效的 Thrift 字段 ID。出于完整性的考虑，这里提供了类的完整代码清单，但重点是集合的创建和该集合执行 FieldStructure 功能的作用。

代码清单 5-8 PropertyStructure 类

```

protected static class PropertyStructure implements FieldStructure {
    private TFieldIdEnum valueId;
    private HashSet<Short> validIds;

    public PropertyStructure(Class prop) {
        try {
            Map<TFieldIdEnum, FieldMetaData> propMeta = getMetadataMap(prop);
            Class valClass = Class.forName(prop.getName() + "Value");
            valueId = getIdForClass(propMeta, valClass);

            validIds = new HashSet<Short>();
            Map<TFieldIdEnum, FieldMetaData> valMeta
                = getMetadataMap(valClass);
            for(TFieldIdEnum valId: valMeta.keySet()) {
                validIds.add(valId.getThriftFieldId());
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public boolean isValidTarget(String[] dirs) {
        if(dirs.length < 2) return false;
        try { 1((check))
            short s = Short.parseShort(dirs[1]);
            return validIds.contains(s);
        } catch (NumberFormatException e) {
            return false;
        }
    }

    public void fillTarget(List<String> ret, Object val) {
        ret.add("'" + ((TUnion) ((TBase)val)
            .getFieldValue(valueId))
            .getSetField()
            .getThriftFieldId());
    }

    private static Map<TFieldIdEnum, FieldMetaData>
        getMetadataMap(Class c)
    {
        try {
            Object o = c.newInstance();
            return (Map) c.getField("metaDataMap").get(o);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private static TFieldIdEnum getIdForClass(
        Map<TFieldIdEnum, FieldMetaData> meta, Class toFind)
    {
        for(TFieldIdEnum k: meta.keySet()) {

```

属性值类型的 Thrift ID 的集合。

Property 是包含属性值字段的 Thrift 结构体, 是这个字段的 ID。

解析 Thrift 元数据, 以获取属性值的字段 ID。

解析 Thrift 元数据, 以获取属性值的所有合法字段 ID。

属性值垂直分区的深度至少为两层。

使用 Thrift ID 来创建当前记录的目录路径。

getMetadataMap 和 getIdForClass 方法是检查 Thrift 对象的辅助函数。


```

FieldValueMetaData md = meta.get(k).valueMetaData;
if(md instanceof StructMetaData) {
    if(toFind.equals(((StructMetaData) md).structClass)) {
        return k;
    }
}
}

throw new RuntimeException("Could not find " + toFind.toString() +
    " in " + meta.toString());
}

```

在这段代码后，休息一下吧——这是你应得的。好消息是，这是一劳永逸的，一旦为主数据集定义了一个 Pail 结构，那么以后与批处理层的交互将会很简单。此外，这段代码适用于任何已创建了 Thrift 图模式的项目。

5.4 总结

你了解了 HDFS 中数据集的维护，包括追加新数据到主数据集、垂直分区数据到多个文件夹和合并小文件等常见任务，并了解了直接使用 HDFS API 完成这些任务是乏味且容易出现人为错误的。

本章介绍了 Pail 抽象。Pail 将你与 HDFS 的文件格式和目录结构隔离开，使其容易实现鲁棒性、执行垂直分区和数据集上一些常见的操作。使用 Pail 抽象最终只需要很少的代码行。垂直分区是自动发生的，并且像追加与合并等任务一样，都是简单的一行代码。这意味着你可以专注于想要对记录进行的何种处理，而不是如何存储这些记录的细节。

有了 HDFS 和 Pail，我们提出了一种存储主数据集的方式，能够满足所有需求，且使用起来很简洁。不管你是否选择使用这些工具，我们希望已为架构的简洁设定了一个标准，而且你的目标至少是实现相同级别的简洁性。

在第 6 章中，你将了解如何利用记录存储来完成 Lambda 架构的下一个关键步骤——计算批处理视图。

批处理层

本章内容

- ❑ 批处理层上的计算方法
- ❑ 将查询分成预先计算和动态组件两部分
- ❑ 重新计算与增量算法
- ❑ 可扩展性的意义
- ❑ MapReduce 范式
- ❑ 一种关于 MapReduce 的更高级的思维方式

数据系统的目标是回答关于数据的任何问题。任何询问数据集的问题都能以一个方法实现，该方法将所有数据作为输入。在理想情况下，无论何时查询数据集，你都可以动态运行这些方法。不幸的是，使用整个数据集作为输入的方法需要运行很长时间。如果你想快速得出查询结果，那么就需要一个不同的策略。

在 Lambda 架构中，批处理层预先计算主数据集到批处理视图中，以便查询可以被低延迟地响应。这需要在预计算什么内容和执行时期计算什么内容之间达成平衡，以完成查询。通过做一些动态计算来完成查询，可以省去预先计算大量批处理视图的操作——关键是要预先计算足够的信息，以便查询可以快速完成。

在前两章中，你学习了如何为数据集建立数据模型，以及如何以可扩展的方式在批处理层中存储数据。在本章中，你将进行下一步的学习，即如何在数据上运行任意的方法。

首先介绍一些我们使用的具有启发性的示例，以说明批处理层计算的概念。其次你将详细学习如何计算主数据集的索引——应用程序层将用它来完成查询。你将检查在重新计算算法、批处理层中强调的算法风格和增量算法以及常用于关系型数据库的一类算法之间的取舍。你将知道批处理层可扩展意味着什么，然后你将学习 MapReduce——它是可扩展且几乎能完成任意批处理计算的范式。你将明白，虽然 MapReduce 是一项伟大的原生框架，但它是很低级的抽象。我们将通过用 MapReduce 执行更高级别的范式来完成任务。

6.1 启发性示例

让我们思考一些查询示例来推动本章理论性的讨论。这些查询说明了批处理计算的概念——每个示例展示了把整个主数据集作为输入时如何将查询看作计算方法。稍后你将使用预先计算来修改这些实现，而不是完全动态地执行查询。

6.1.1 给定时间范围内的页面浏览量

第一个查询示例运行在页面浏览量的数据集上，其中每条页面浏览记录包含了 URL 和时间戳。查询的目的是确定一个 URL 在给定时间范围内的总页面浏览量。

该查询可以写成如下所示的伪代码：

```
function pageviewsOverTime(masterDataset, url, startHour, endHour) {
    pageviews = 0
    for(record in masterDataset) {
        if(record.url == url &&
            record.time >= startHour &&
            record.time <= endHour) {
            pageviews += 1
        }
    }
    return pageviews
}
```

使用整个数据集的方法来运行该查询，你只需要简单地遍历每条记录，并为指定时间范围内的该 URL 的页面浏览量保留一个计数器。在遍历完所有记录之后，系统将返回计数器的最终值。

6.1.2 性别推理

下一个查询示例运行在名字记录的数据集上，用以预测一个人可能的性别。该算法首先对人的名字执行语义规范化，做类似 Bob 到 Robert、Bill 到 William 的转换。该算法使用

一个为每个名字提供性别概率的模型。

由此产生的推理算法如下所示：

```

规范化所有人名。
function genderInference(masterDataset, personId) {
    names = new Set()
    for(record in masterDataset) {
        if(record.personId == personId) {
            names.add(normalizeName(record.name))
        }
    }
    maleProbSum = 0.0
    for(name in names) {
        maleProbSum += maleProbabilityOfName(name)
    }
    maleProb = maleProbSum / names.size()
    if(maleProb > 0.5) {
        return "male"
    } else {
        return "female"
    }
}

```

每个名字是男性的概率的平均值。

返回性别的最高似然估计结果。

该查询的有趣之处在于，随着时间的推移，名字规范化算法和名字 - 性别模型的改善，查询结果是可以改变的，而不仅仅是在接收到新数据时才发生改变。

6.1.3 影响力分数

最后一个示例运行在 Twitter 授权的包含反应记录的数据集上。每条反应记录包含 sourceId 和 responderId 字段，分别表示 responderId 转发或回复 sourceId 的帖子。

该查询确定了社交网络中每个人的影响力分数。该分数通过两个步骤计算得到：首先，每个人的首要影响者是基于影响者造成这个人反应的总数来选择的；其次，每个人的影响力分数由他是多少人的首要影响者的数量来确定。

确定用户影响力分数的算法如下：

```

function influence_score(masterDataset, personId) {
    influence = new Map()
    for(record in masterDataset) {
        curr = influence.get(record.responderId) || new Map(default=0)
        curr[record.sourceId] += 1
        influence.set(record.sourceId, curr)
    }
    score = 0
    for(entry in influence) {
        if(topKey(entry.value) == personId) {
            score += 1
        }
    }
}

```

计算所有成对的人之间的影响力。

对 personId 是首要影响者的人进行计数。


```
} 一些我们使用的具有自发性的示例，以说明批处理层如何对查询中各个部分进行  
return score  
}
```

在这段代码中，topKey 方法是虚拟的（即只是调用，实际上并没有实现），因为它的实现很简单。否则，算法只是简单地计算每对人之间反应的数量，然后对查询用户是首要影响者的人计数。

6.2 批处理层上的计算

让我们回顾一下 Lambda 架构在较高层次上是如何工作的。在处理查询时，Lambda 架构的各层都有一个关键的互补的角色，如图 6-1 所示。

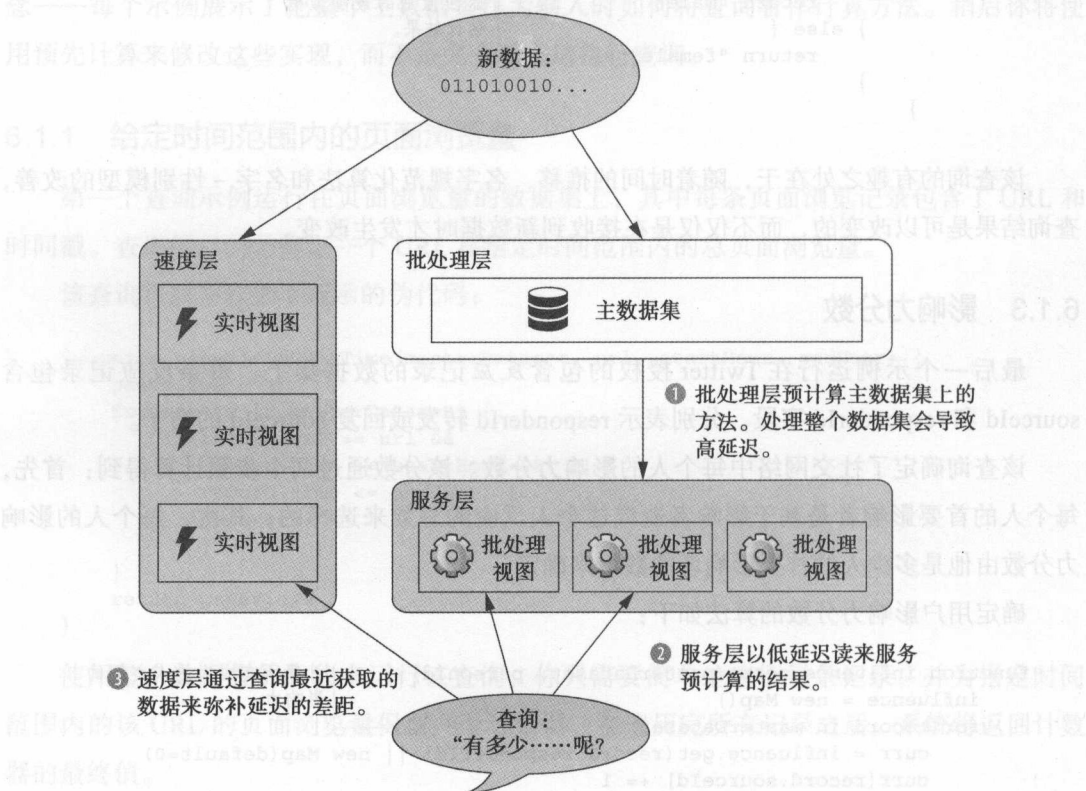


图 6-1 在服务数据集查询时 Lambda 架构各层的角色

批处理层在主数据集上运行方法，来预先计算被称为批处理视图的中间数据。批处理视图由服务层加载和索引，使得批处理视图能够快速访问数据。速度层通过使用尚未预先计算到批处理视图的数据进行更新来提供低延迟，从而弥补了批处理层的高延迟；然后处

理来自服务层视图和速度层视图的数据，并合并结果，以满足查询。

该架构的关键是对于任何查询，都可以通过预先计算批处理层的数据来加快服务层的处理。这些在主数据集上的预先计算需要花费时间，但是你应该将批处理层的高延迟看作深入分析数据和连接不同数据的机会。记住，低延迟查询服务是通过 Lambda 架构的其他部分实现的。

批处理层上计算的一个简单策略是，预先计算所有可能的查询，并将结果缓存在服务层中。这种方法的示意图如图 6-2 所示。

遗憾的是，你不能总是预先计算一切查询。以给定时间范围内的页面浏览量查询为例，如果你想预先计算每一个可能的查询，就需要确定每个 URL、每个可能时间范围的答案。但是给定时间框架内小时范围的数量是巨大的。在一年中，大约有 3.8 亿种不同的小时范围。为了预先计算查询，你需要预先计算和索引每个 URL 的 3.8 亿个值，这显然是不可行且不切实际的解决方案。

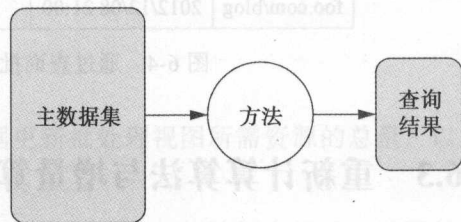


图 6-2 通过在主数据集上直接运行方法来预先计算查询

相反，你可以预先计算中间结果，然后用这些结果来动态完成查询，如图 6-3 所示。

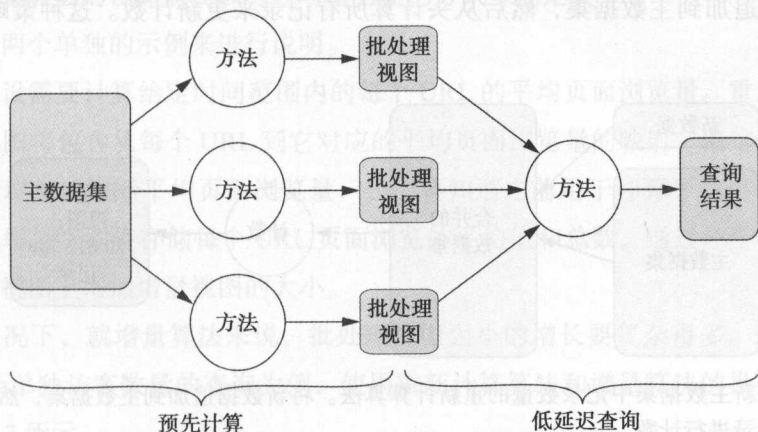


图 6-3 将查询分成预先计算和动态组件

对于给定时间范围内的页面浏览量查询，你可以预先计算每个 URL 每个小时的页面浏览量，如图 6-4 所示。

要完成一个查询，你需要从索引中获得该查询的时间范围内每小时的页面浏览量，并将结果进行累加。若时间范围为一年，则对于每个 URL，你只需要预先计算并索引 8760 个值（365 天，每天 24 小时）。这显然是更易于管理的数量。

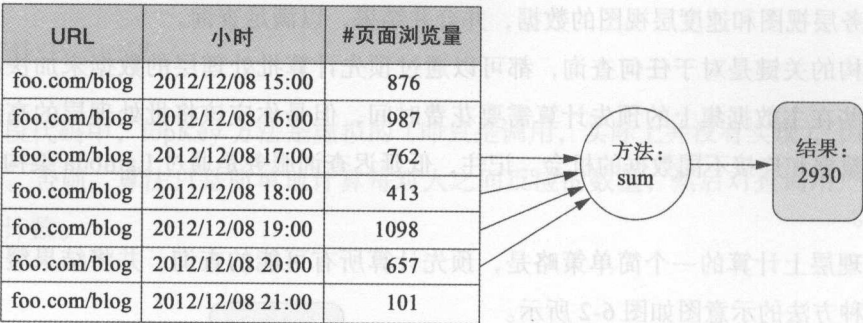


图 6-4 通过查询批处理视图的索引来计算页面浏览量

6.3 重新计算算法与增量算法

因为主数据集是不断增长的，所以当新数据可用时，你必须有一个更新批处理视图的策略。一种方法是，选择重新计算算法，丢弃旧的批处理视图，重新计算整个主数据集上的方法；另一种方法是，当新数据到达时，使用增量算法直接更新视图。

举一个基本的示例，假设一个批处理视图包含主数据集中记录的总数。重新计算算法首先将新数据追加到主数据集，然后从头计算所有记录来更新计数。这种策略的示意图如图 6-5 所示。

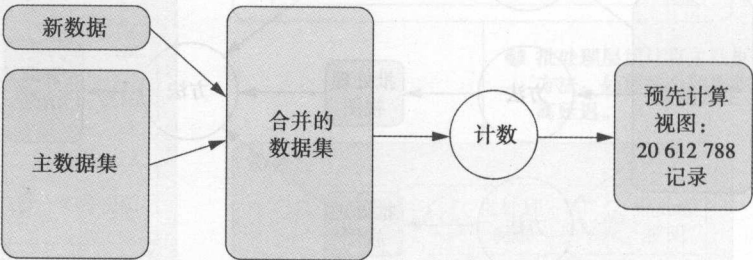


图 6-5 更新主数据集中记录数量的重新计算算法。将新数据追加到主数据集，然后对所有记录进行计数

而另一方面，增量算法将计算新数据记录的数量，并将其添加到现有的计数中，如图 6-6 所示。

你可能想知道，在可以使用更高效的增量算法时，为什么曾经会使用重新计算算法。这是因为效率并不是唯一要考虑的因素。这两种方法的关键取舍之处在于性能、容忍人为错误以及算法的通用性。我们将依次就这些方面来讨论这两种算法。你会发现，尽管增量算法可以提供额外的效率，你也一定需要重新计算算法。

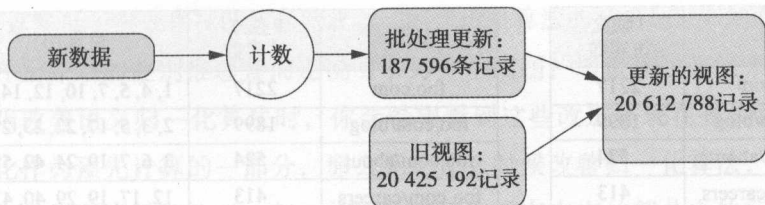


图 6-6 更新主数据集中记录数量的增量算法。只对新数据集计数，再加上总数来直接更新批处理视图

6.3.1 性能

批处理层算法的性能表现在两个方面：用新数据更新批处理视图所需资源的总量，以及所生成批处理视图的大小。

增量算法几乎总是用明显更少的资源来更新视图，因为它使用新数据和批处理视图的当前状态来执行更新。对于计算给定时间范围内的页面浏览量等任务，由于聚合，视图将明显小于主数据集。重新计算算法着眼于整个主数据集，因此更新所需的资源比增量算法高出多个数量级。但增量算法的批处理视图的大小明显大于对应的重新计算算法的批处理视图，因为增量算法的视图需要以可以增量更新的方式来制定。

下面通过两个单独的示例来进行说明。

首先，假设需要计算给定时间范围内的每个 URL 的平均页面浏览量。重新计算算法生成的批处理视图将包含从每个 URL 到它对应的平均页面浏览量的映射，但这并不适用于增量算法，因为增量地更新平均页面浏览量，还需要知道之前用于计算平均页面浏览量的记录总数。因此增量视图将存储每个 URL 页面浏览的平均值和总数，通过一个常数因子在基于重新计算的视图上增加增量视图的大小。

在其他情况下，就增量算法来说，批处理视图大小的增长要复杂得多。以一个为每个 URL 计算每个单独访客数量的查询为例，使用重新计算算法和增量算法的批处理视图之间的差异如图 6-7 所示。

重新计算算法的批处理视图只需要一个从 URL 到独立访客计数的映射。相比之下，增量算法只检查新的页面浏览量，所以它的视图必须要包含每个 URL 的访客的全部集合，这样才可以确定对应新数据中的记录，进而返回访客计数。因此，增量算法的视图可能和主数据集一样大！

使用增量算法生成的批处理视图并不总是这么大，但它远远大于对应的基于重新计算算法的批处理视图。

URL	#独立 访客数	URL	#独立 访客数	访客ID
foo.com	2217	foo.com	2217	1, 4, 5, 7, 10, 12, 14, ...
foo.com/blog	1899	foo.com/blog	1899	2, 3, 5, 17, 22, 23, 27, ...
foo.com/about	524	foo.com/about	524	3, 6, 7, 19, 24, 42, 51, ...
foo.com/careers	413	foo.com/careers	413	12, 17, 19, 29, 40, 42, ...
foo.com/faq	1212	foo.com/faq	1212	8, 10, 21, 37, 39, 46, 55, ...
...

重新计算算法的批处理视图

增量算法的批处理视图

图 6-7 确定每个 URL 的每个单独访客数量的重新计算算法和增量算法的批处理视图之间的差异

6.3.2 容忍人为错误

数据系统的生命周期非常长，在该周期内程序缺陷也可能被部署到生产中，因此你必须考虑批处理更新算法如何容忍这种错误。在这方面，重新计算算法能够容忍人为错误；而对于增量算法，人为错误可能会导致严重的问题。

现有一个批处理层算法的示例，用以计算主数据集中记录数量的全局计数。现在假设有一个人为错误——你部署了一个为每个记录的全局计数增加 2 而不是 1 的算法。如果该算法是基于重新计算的，那么只需要修复算法并重新部署代码——这样在下次批处理层运行时，批处理视图将被修正。这是因为基于重新计算的算法会从头计算批处理视图。

但是如果该算法是增量的，那么修正视图就不会那么简单了。唯一的选择是鉴别出计数过多的记录，进而确定每个记录被多计数了几次，然后修正每个受影响的记录的计数，但并不能保证每次都能完整、准确地完成这项任务。详细的日志记录可能有助于完成这些任务，但日志可能并不总是包含必要的信息，因为你无法预料错误的每一种类型。很多时候，你不得不对视图做临时的、最佳猜测的修改——同时也必须确保没把它们搞得一团糟。

希望有正确的日志来修复错误并不是良好的工程实践。需要重复的是：人为错误是不可避免的。如你所见，比起增量算法，基于重新计算的算法更能容忍人为错误。

6.3.3 算法的通用性

尽管增量算法可以更快地运行，但它们常常需要定制才能解决问题。例如，你已经知道了计算独立访客数量的增量算法可以生成非常大的批处理视图，这个开销可以通过像 HyperLogLog 这样的概率统计算法抵消——HyperLogLog 存储中间数据来估计总体的独立

访客计数^①。这降低了批处理视图的存储成本，但代价是算法的结果是近似的而不是准确的。

本章刚开始介绍的性别推理查询还说明了另一个问题：增量算法将复杂性转移到了动态计算中。当改善语义归一化算法时，你会希望看到这些改进体现在查询结果中。但是，如果将归一化作为预先计算的一部分，那么无论什么时候改善归一化算法，批处理视图都会是过时的。当使用增量算法时，归一化算法必须在动态查询的部分中执行。批处理视图必须包含每个出现过的人的姓名，并且每执行一次查询，动态代码将不得不重新归一化每个姓名。这就增加了动态组件的延迟，很可能需要花费很长时间来满足应用程序的需求。

因为重新计算算法不断重建整个批处理视图，所以批处理视图的结构和动态组件的复杂性都会更简单，更容易产生通用的算法。

6.3.4 选择算法的风格

表 6-1 所示为重新计算算法和增量算法的对比。

表 6-1 重新计算算法和增量算法的对比

	重新计算算法	增量算法
性能	需要处理整个主数据集的计算工作量	需要更少的计算资源但可能产生大得多的批处理视图
容忍人为错误	非常容忍人为错误，因为批处理视图被不断重建	不容易修复批处理视图中的错误；修复是暂时的，并可能需要估算
通用性	在预先计算阶段解决了算法的复杂性，由此生成了简单的批处理视图和低延迟、动态处理	需要特殊定制；可能将复杂性转移到动态查询的处理中
总结	对于支持鲁棒性的数据处理系统是必不可少的	可以提高系统的效率，但只是作为重新计算算法的一种补充

关键在于，你必须有一直重新计算算法的版本，这是确保系统能够容忍人为错误的唯一方法，并且容忍人为错误对一个具有鲁棒性的系统来说，是不可或缺的需求。此外，你可以选择为算法添加增量版本，使其具有更高的效率。

在本章的其余部分，我们将只关注重新计算算法；但是在第 18 章中，我们将再次回到增量批处理层这个主题。

6.4 批处理层中的可扩展性

围绕可扩展性这个词可以引申出很多内容，所以你应该仔细定义它在数据系统背景中的

^① 我们将在后续章节中进一步讨论 HyperLogLog。

含义。可扩展性是一个系统通过添加更多的资源，在负载增加的情况下维持性能的能力。在大数据背景下，负载是数据总量、每天收到的新数据量、应用程序每秒服务的请求数等数据的组合。

比系统可扩展更重要的是，系统是线性可扩展的。通过添加与增加的负载成正比的资源，线性可扩展的系统在负载增加的情况下，仍能维持性能。对于非线性可扩展的系统来说，尽管它是“可扩展的”，但它并不是特别有用。假设所需要的机器数量与系统上的负载成平方关系，如图 6-8 所示，随着时间的推移，系统运行的成本将大幅上升。负载增加 10 倍，那么成本将会增加 100 倍。从成本角度来看，这样的系统是不可行的。

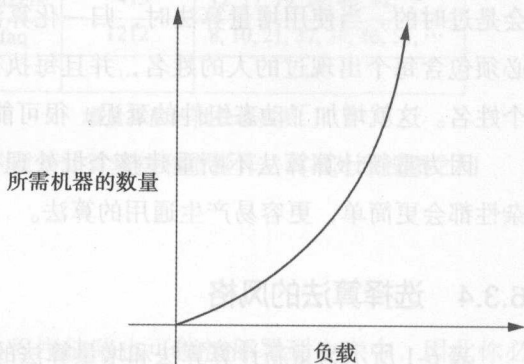


图 6-8 非线性可扩展性

如果一个系统是线性可扩展的，那么成本与负载成正比——这是数据系统的一个至关重要的属性。

可扩展性并不意味着……

相反，可扩展的系统并不一定通过添加更多的机器就能够提升性能。举一个相关的例子，假设有一个能提供静态 HTML 页面服务的网站，每个 Web 服务器在 100ms 的延时要求内可以提供每秒 1000 个请求的服务。你 cannot 通过添加更多的机器来降低服务 Web 页面的延迟——一个单独的请求是不能并行的，并且必须由一台机器来满足。但是你可以通过添加更多 Web 服务器分散服务 HTML 的负载来扩展网站，以增加每秒的请求数量。

更实际的是，因为算法是可并行的，你可以通过添加更多的机器来提升性能，但随着机器的增多，改善效果将会减弱，因为更多的机器会增加开销和通信成本。

我们深入探讨了可扩展性，是为了给介绍 MapReduce 做好准备。MapReduce 是一种分布式计算模式，可以用来实现批处理层。在讨论工作细节时，要记住它是线性可扩展的——如果你将主数据集的大小翻倍，那么两倍的服务器数量就能够以相同的延迟创建批处理视图。

6.5 MapReduce：一种大数据计算的范式

MapReduce 是最初由 Google 研发的一种分布式计算模式，提供可扩展性和批处理计算容错性的原语。利用 MapReduce，可以将计算写成关于 map 和 reduce 的方法来操作键/值对。这些原语对于实现几乎任何方法都是极具表现力的，并且 MapReduce 框架以一种分

布式和具有鲁棒性的方式在主数据集上执行这些方法。这些属性使得 MapReduce 成为批处理层预先计算算法需要的一种优秀范式，但它也是一类低层次的抽象，它所表达的计算需要很大的工作量。

典型的 MapReduce 示例是 word count。Word count 用于获得一个文本的数据集，并确定每个单词在整个文本中出现的次数。MapReduce 中的 map 方法对每行文本执行一次并生成任意数量的键 / 值对。对于 word count，map 方法为文本中的每个单词生成一个键 / 值对，设置键为单词，且值为 1：

```
function word_count_map(sentence) {
  for(word in sentence.split(" ")) {
    emit(word, 1)
  }
}
```

接着 MapReduce 整理 map 方法的输出，使得相同键的所有值被分在一组。

然后 reduce 方法获取共享相同键的值的完整列表，并生成新的键 / 值对作为最终的输出。在 word count 中，输入是每个单词值为 1 的一个列表，reducer 只是将值相加来计算每个单词的计数：

```
function word_count_reduce(word, values) {
  sum = 0
  for(val in values) {
    sum += val
  }
  emit(word, sum)
}
```

在机器的集群上运行一个像 word count 这样的程序，运行后台会发生很多事情，但 MapReduce 框架处理了大部分的细节，目的是让用户关注需要计算什么而不用关心计算的细节。

6.5.1 可扩展性

MapReduce 是如此强大的一个范式的原因是，依据 MapReduce 编写的程序本质上是可扩展的。一个运行在 10 GB 数据上的程序也可以运行在 10 PB 的数据上。无论输入大小，MapReduce 都在机器的集群上自动地并行计算。并发性、机器之间的数据传输和执行计划的所有细节对用户来说都是抽象的，它们由 MapReduce 框架处理。

下面来看诸如 word count 这样的程序是如何在 MapReduce 集群上执行的。MapReduce 程序的输入是存储在分布式文件系统上的，比如第 5 章提到的 Hadoop 分布式文件系统 (HDFS)。在处理数据之前，程序首先确定集群的哪些机器托管包含输入数据的块，如

图 6-9 所示。

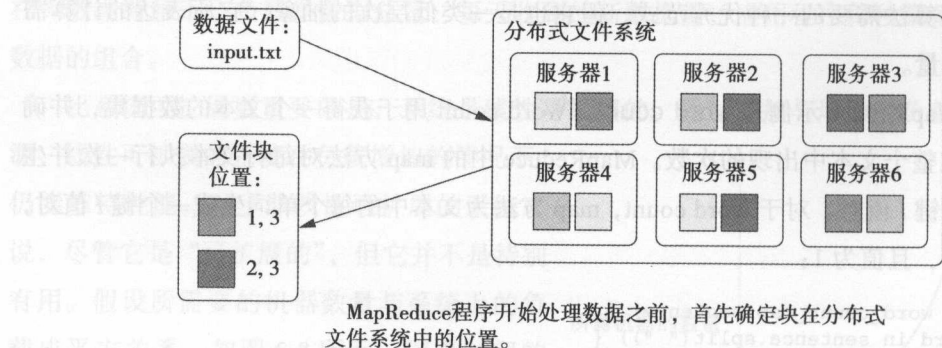


图 6-9 定位托管 MapReduce 程序输入文件的服务器

确定输入的位置之后, MapReduce 启动一些与输入数据大小成正比的 map 任务数量。每个任务被分配了一个输入子集, 并在该数据上执行 map 方法。因为代码的总量通常远远低于数据总量, 所以 MapReduce 试图将任务分配给托管被处理数据的服务器, 如图 6-10 所示, 将代码移动到数据所在的服务器, 避免了在网络上传输所有数据。

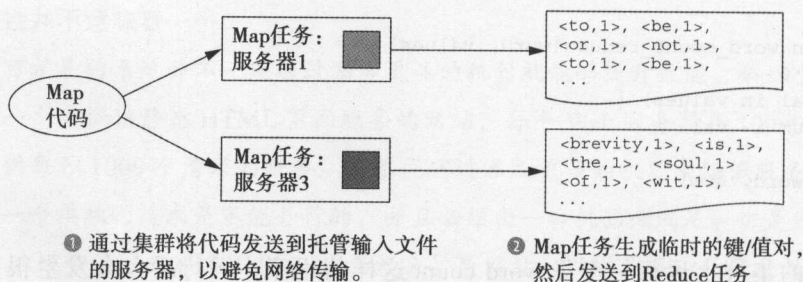
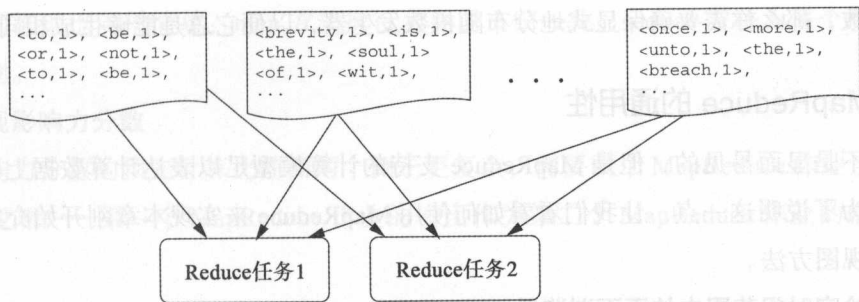


图 6-10 MapReduce 促进数据本地化, 在输入数据所在服务器上运行任务

与 map 任务类似, reduce 任务也分布在集群中。每个任务负责运行由 map 任务生成的键的子集的 reduce 方法。因为 reduce 方法需要用到与给定键相关的所有值, 所以直到所有 map 任务完成, reduce 任务才会开始。

一旦 map 任务结束执行, 每个生成的键 / 值对将被发送给负责处理该键的 reduce 任务。因此, 每个 map 任务将其输出分配给所有 reducer 的任务。这种中间键 / 值对的传输被称为洗牌 (shuffle), 如图 6-11 所示。

一旦 reduce 任务接收到来自每个 map 任务的所有键 / 值对, 它会按照键的顺序对键 / 值对进行排序 (这对将给定键的所有值组织在一起有影响), 然后对每个键和该键的所有值执行 reduce 方法, 如图 6-12 所示。



在洗牌阶段，Map任务生成的所有键/值对分发到Reduce任务上。
在这个过程中，相同键的所有对被发送到相同的reducer上。

图 6-11 洗牌过程将 map 任务的输出分配到 reduce 任务中

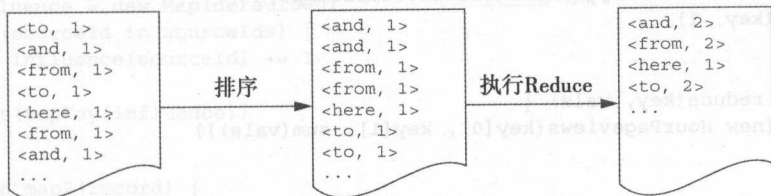


图 6-12 reduce 任务根据键来对输入数据排序，然后对生成的所有值执行 reduce 方法。

如你所见，一个 MapReduce 程序有很多值得关注的部分。重要的概述如下：

- ❑ MapReduce 程序以完全分布式的方式执行，这没有争议。
- ❑ MapReduce 是可扩展的：所提供的 map 和 reduce 方法是跨集群并行执行的。
- ❑ 为你处理了并发性和给机器分配任务的挑战。

6.5.2 容错性

分布式系统是出了名的易出错。对于单台服务器，网络分区、服务器崩溃和磁盘故障相对少见，但是在大型机器集群上协调计算时，出错的可能性就大大增加。值得庆幸的是，除了易于并行化和固有的可扩展性之外，MapReduce 计算也具有容错性。

程序可能因为各种原因而失败：硬盘达到饱和、进程超过可以执行的可用内存或者硬件发生故障。MapReduce 监视这些错误，并自动在另一个节点上重试该部分的计算。只有当一个任务失败的次数超过配置数量——通常是 4 次，整个应用程序（通常称为作业）才会失败。一般认为，单次失败可能是服务器的问题，重复的失败则可能是代码的问题。

因为任务可以重试，所以 MapReduce 要求 map 和 reduce 方法是确定的。这意味着给定相同的输入，方法必须生成相同的输出。这是一个相对宽松的约束，但对 MapReduce 正常工作是很重要的。非确定性方法的一个例子是产生随机数。如果想在 MapReduce 作业中

使用随机数，那么你需要确保显式地分布随机数发生器，以便它总是能够生成相同的输出。

6.5.3 MapReduce 的通用性

虽然不是显而易见的，但是 MapReduce 支持的计算模型足以表达计算数据上的几乎所有方法。为了说明这一点，让我们看看如何使用 MapReduce 来实现本章刚开始介绍的查询的批处理视图方法。

实现给定时间范围内的页面浏览量

下面的 MapReduce 代码可以生成给定时间范围内的页面浏览量的批处理视图：

```
function map(record) {
    key = [record.url, toHour(record.timestamp)]
    emit(key, 1)
}

function reduce(key, vals) {
    emit(new HourPageviews(key[0], key[1], sum(vals)))
}
```

这段代码与 word count 代码非常类似，但 mapper 生成的键是一个包含 URL 和页面浏览的小时值的结构体；reducer 的输出是所需的批处理视图，包含一个从 [url, hour] 到该小时的页面浏览量的映射。

实现性别推理

下面的 MapReduce 代码可以推断所提供的名字的性别：

```
function map(record) {
    emit(record.userid, normalizeName(record.name))
}
```

语义归一化发生在 mapping 阶段。

```
function reduce(userid, vals) {
    allNames = new Set()
    for(normalizedName in vals) {
        allNames.add(normalizedName)
    }
    maleProbSum = 0.0
    for(name in allNames) {
        maleProbSum += maleProbabilityOfName(name)
    }
    maleProb = maleProbSum / allNames.size()
    if(maleProb > 0.5) {
        gender = "male"
    } else {
        gender = "female"
    }
    emit(new InferredGender(userid, gender))
}
```

一个用于移除任何可能重复的名称的集合。

男性概率的平均值。

返回最可能的性别。

性别推理也同样简单：map 方法执行名称语义归一化，reduce 方法为每个用户计算预测的性别。

实现影响力分数

影响力分数的预先计算比前面两个例子更复杂，需要两个 MapReduce 作业连接在一起来实现逻辑——第一个 MapReduce 作业的输出作为第二个 MapReduce 作业的输入。代码如下：

```
function map1(record) {
    emit(record.responderId, record.sourceId)
}

function reduce1(userid, sourceIds) {
    influence = new Map(default=0)
    for(sourceId in sourceIds) {
        influence[sourceId] += 1
    }
    emit(topKey(influence))
}

function map2(record) {
    emit(record, 1)
}

function reduce2(influencer, vals) {
    emit(new InfluenceScore(influencer, sum(vals)))
}
```

第一个作业确定了每个用户的首要影响者。

然后首要影响者数据用来确定每个用户影响的人的数量。

这是典型的需要多个 MapReduce 作业的计算——这仅仅意味着需要多级分组。这里的第一个作业需要对每个用户的反应进行分组，以确定该用户的首要影响者；第二个作业接着根据首要影响者对记录进行分组，以确定影响力分数。

退一步来看，MapReduce 在基础层级做了什么：

- ❑ map 阶段将数据根据键值进行任意的数据分区。在稍后的处理中，任意分区使系统能将数据连接在一起，同时仍然能并行处理一切。
- ❑ 在 map 和 reduce 阶段提供的代码能够任意转换数据。

很难想象有任何其他更通用的可扩展的、分布式的系统。

MapReduce 和 Spark

Spark 是一个已经获得了很多关注、相对较新的计算系统。Spark 的计算模型是“弹性分布式数据集”。与 MapReduce 相比，Spark 不具备更好的通用性或可扩展性，但它的模型在计算时具有更高的性能，可使算法反复在相同的数据集上迭代（因为 Spark 能够在内存中缓存数据，而不是每次都从磁盘读取）。许多机器学习算法反复在相同的数据集上迭代——Spark 特别适用于这些情况。

6.6 MapReduce 的底层特性

不幸的是，尽管 MapReduce 对批处理计算来说是一个强大的原语——提供一个通用、可扩展和容错的方式来计算大型数据集上的方法——但它不适合生成特别优雅的代码。你会发现 MapReduce 手动编写的程序往往冗长、笨拙且难以理解。下面讨论这一特性的原因。

6.6.1 多步计算很怪异

在影响力分数的示例中，计算过程需要两个 MapReduce 作业来完成。所给出代码缺少的部分是将这两个作业连接在一起的代码。运行 MapReduce 作业需要的不仅仅是 mapper 和 reducer——还需要知道在哪里读取它的输入，在哪里写它的输出。这就很麻烦——为了使代码能够运行，你需要在步骤 1 和步骤 2 之间找一个位置来存放中间输出，然后需要清理中间输出，以防它在非必要时间占用宝贵的磁盘空间。

这应该立即引起重视，因为它清楚地表明，系统工作在低层次的抽象中，而你希望整个计算可以表示为单个概念单元的抽象，并且系统自动处理类似管理临时路径的细节。

6.6.2 手动实现连接非常复杂

下面来看一个更加复杂的示例——通过 MapReduce 实现连接。假设有两个单独的数据集：一个包含字段 id 和年龄（age）的记录；另一个包含字段 user_id、性别（gender）和位置（location）的记录。你想计算同时存在于这两个数据集中的每一个 ID 的年龄、性别和位置。这个操作称为内连接，如图 6-13 所示。连接是非常常见的操作，你可能已经通过 SQL 之类的工具熟悉了它。

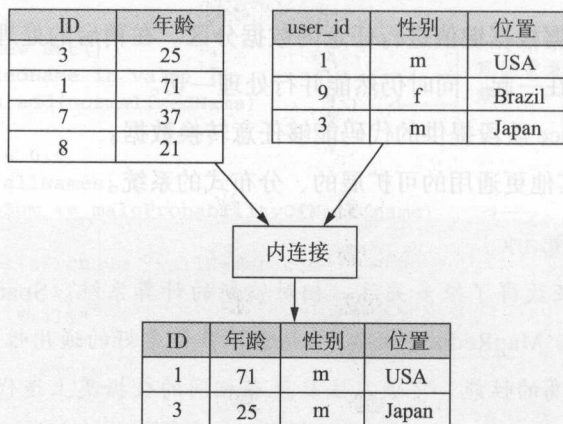


图 6-13 两边内连接的示例

通过 MapReduce 实现连接, 需要在 MapReduce 作业中读取两个独立的数据集, 因此, 作业需要能够区分来自两个数据集的记录。虽然到目前为止, 我们还没有在伪代码中展示过该部分, 但 MapReduce 框架通常提供上下文来标记记录的来源, 因此我们将扩展伪代码来包含该上下文。下面的代码实现了内连接:

```
function join_map(sourcedir, record) {
  if(sourcedir=="/data/age") {
    emit(record.id, {"side" = "l"
    , "values" = [record.age]})
  } else {
    emit(record.user_id,
      {"side" = "r",
      "values" = [record.gender, record.location]})
  }
}

function join_reduce(id, records) {
  side_l = []
  side_r = []

  for(record : records) {
    values = record.get("values")
    if(record.get("side") == "l") {
      side_l.add(values)
    } else {
      side_r.add(values)
    }
  }

  for(l : side_l) {
    for(r : side_r) {
      emit(concat([id], l, r), null)
    }
  }
}
```

使用记录的源目录来确定记录是连接的左边还是右边。

分别将 MapReduce 的键设置为 id 或 user_id。这将使得该连接任一侧的这些 id (或 user_id) 的所有记录进入相同的 reduce 调用中。如果一次连接多个键, 应该将 MapReduce 的键设置为一个集合。

想要放在输出记录中的值先放在该处的列表中。之后它们将与另一侧的记录连接在一起并生成输出。

当进行 reduce 处理时, 首先从连接的任一侧分离记录, 加入“左”和“右”列表中。

id 被添加到连接值中来生成最终的结果。注意: 因为 MapReduce 总是操作键/值对, 在这种情况下, 要将生成的结果作为键, 将值设置为 null (也可以用相反的方式来实现)。

为了实现连接的语义, 将连接的一侧的每个记录与另一侧的每个记录连接在一起。

虽然代码总量不是很大, 但是仍然需要相当多的繁重工作来使机器正常运行。其复杂性是这样的: 确定记录属于连接的哪一边与特定的目录有关, 所以在不同的目录上做连接时必须调整代码。此外, MapReduce 强制所有结果都是键/值对, 这感觉并不适合这个作业的输出, 因为该输出只是一个值的列表。

而这只是一个简单的连接单个字段的两边内连接。试想如果连接多个字段, 且该连接有 5 条边, 一些边作为外连接, 一些边作为内连接, 显然你不想每次都手动编写连接代码, 所以应该能够以更高层次的抽象来实现连接。

6.6.3 逻辑和物理执行紧密耦合

下面来看一个真正确认 MapReduce 是低层次抽象的示例——扩展 word-count 的例子，来过滤出单词 “the” 和 “a”，并生成双倍的计数而不是单倍计数。完成该示例的代码如下：

```
EXCLUDE_WORDS = Set("a", "the")
```

```
function map(sentence) {
  for(word : sentence) {
    if(not EXCLUDE_WORDS.contains(word)) {
      emit(word, 1)
    }
  }
}
```

```
function reduce(word, amounts) {
  result = 0
  for(amt : amounts) {
    result += amt
  }
  emit(result * 2)
}
```

这段代码可以运行，但它似乎将多个任务混合在同一个方法中，而良好的编程实践要求将独立的功能分散到它们各自的方法中。执行该计算更好的思路如图 6-14 所示。

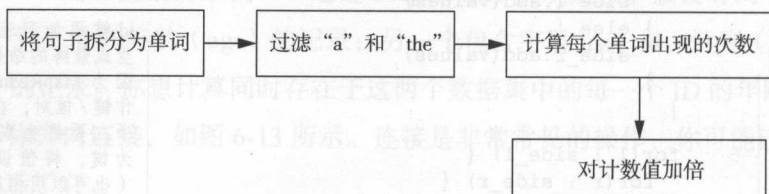


图 6-14 分解修改后的 word-count 问题

你可以分解这段代码，使得每个 MapReduce 作业只完成这些方法的一个功能。但一个 MapReduce 作业意味着一次具体的物理执行：先运行一组 mapper 进程来执行 map 部分，然后会发生磁盘和网络 I/O 以使中间记录进入 reducer，最后运行一组 reducer 进程来产生输出。模块化代码将创建更多不必要的 MapReduce 作业，使得计算效率很低。

所以你要做出一个艰难的取舍——要么是将所有功能组织在一起，这将导致不良的软件工程实践；要么模块化代码，这将导致资源使用情况不佳。但事实上，你根本不用做这个取舍，而应该综合应用两者的优点——将完全模块化的代码编译成最佳物理执行。下面来看如何实现这一点。

6.7 管道图——一种关于批处理计算的高级思维方式

本节将介绍一种关于批处理计算更自然的思维方式——管道图。管道图可以被编译为一系列有效的 MapReduce 作业来执行。正如你将看到的，所展示的每个示例——包括 SuperWebAnalytics.com 的所有内容——都可以用管道图简明地表示。

管道图只是使我们能够在 Lambda 架构中专注于批量计算，而不迷失在 MapReduce 伪代码的细节中。它的关键是简洁性和直观性——这两者都是 MapReduce 所缺乏的，却正是管道图所擅长的。此外，管道图让我们可以专注于所要解决的示例问题的具体算法和数据处理转换，而不陷入具体工具的细节中。

管道图实践

管道图不是假设的概念；所有高级 MapReduce 工具都是管道图的直接映射，包括 Cascading、Pig、Hive 和 Cascalog。在某种程度上 Spark 也是，尽管它的数据模型并无固有的包括任意数量命名字段的元组的概念。

6.7.1 管道图的概念

管道图背后的想法是思考关于元组、方法、过滤器、聚合器、连接和聚合的处理——你可能在 SQL 中已经熟悉了这些概念。例如，图 6-15 所示为 6.6.3 节修改后的带有过滤与双倍加和的 word-count 示例的管道图。

该计算开始于名为“sentence”的单个字段的元组。Split 方法将单个 sentence 元组转换成多个字段为 word 的元组，即 Split 将 sentence 字段作为输入，并创建 word 字段作为输出。

图 6-16 所示为给出一组 sentence 元组，应用 Split 方法后所发生的事情的图解。可以看到，sentence 字段被复制到了所有新元组中。

当然，管道图中的方法并不仅限于预先设定的一组方法，它们可以是通用的编程语言实现的任意方法。这同样适用于过滤器和聚合器。

接下来，使用移除“a”和“the”的过滤器，其产生的效果如图 6-17 所示。

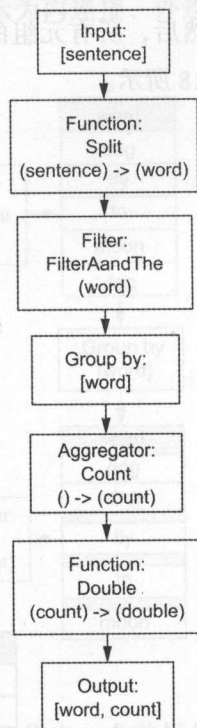


图 6-15 修改后的带有过滤与双倍加和的 word-count 示例的管道图

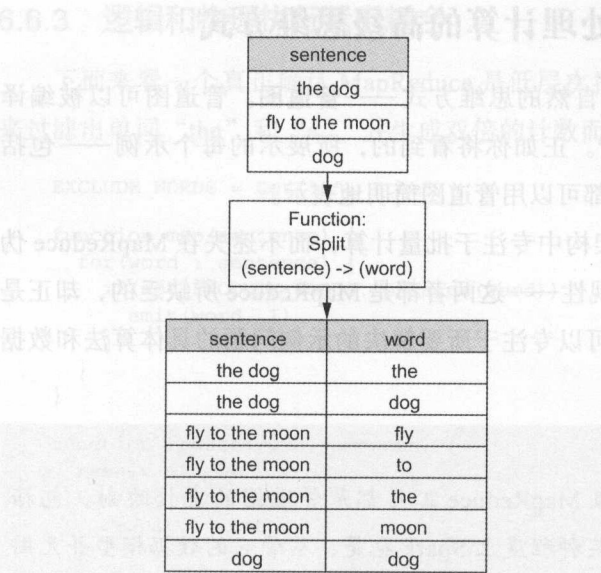


图 6-16 一个管道图方法的图解



图 6-17 管道图过滤器的图解

然后，所有元组的集合根据字段 word 分组，并且将 count 聚合器应用于每个组，如图 6-18 所示。

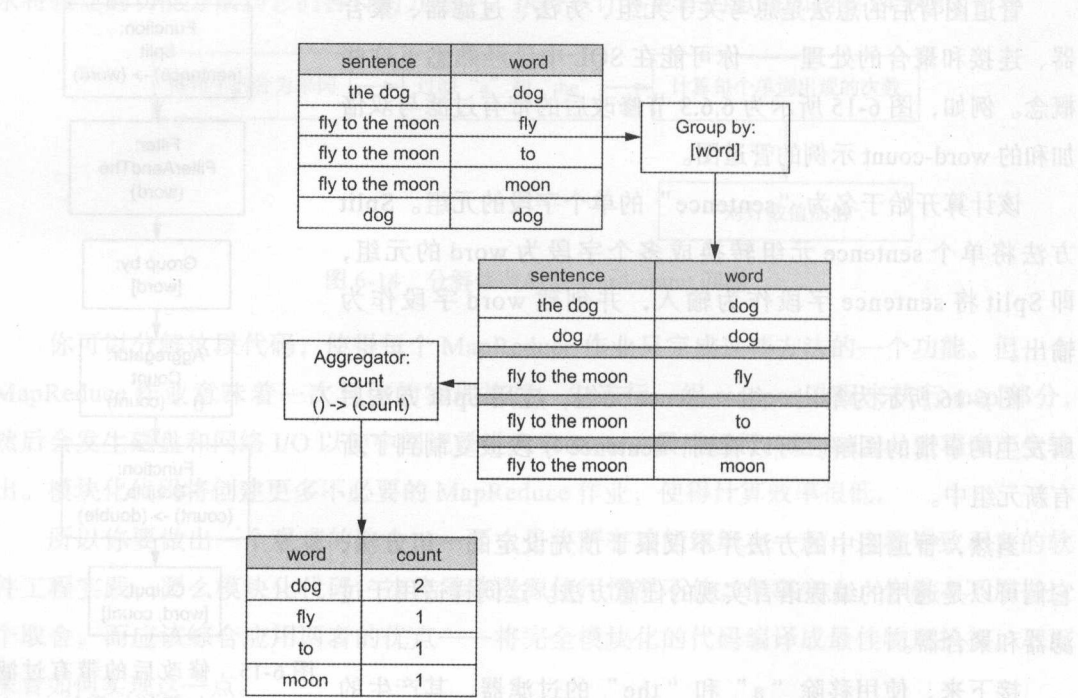


图 6-18 管道图分组与聚合

之后，将 count 翻倍后创建新的字段 double，如图 6-19 所示。

最后，选择最终所需的字段作为输出并舍弃其他字段。

如上所示，管道图的关键点之一是，字段一旦被创建，就不可变。你可以做的一个明显的优化是，只要字段不再被需要就舍弃它们（防止不必要的序列化和网络 I/O）。在大多数情况下，实现管道图的工具会自动做这种优化。所以在现实中，前面的示例将按照图 6-20 所示的步骤执行。

管道图中还有另外两类重要的操作，这两类操作均用于组合独立元组集。

第一类是连接操作，它允许你在任意数量的元组集之间做内连接和外连接。尽管在如何指定每一条边的连接字段方面使用的工具有所差异，但最简单的表示是选择连接的所有边的公共字段作为连接字段。这需要确保想连接的字段都是完全相同的名字。

然后，连接的每一边被标记为“内连接”或“外连接”。图 6-21 所示为内连接、外连接与混合连接示例。

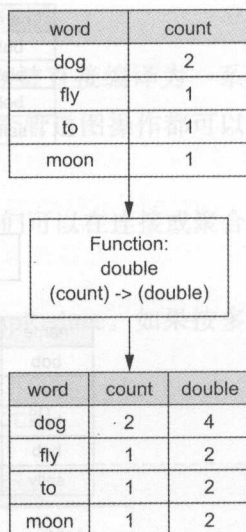


图 6-19 运行 double 方法

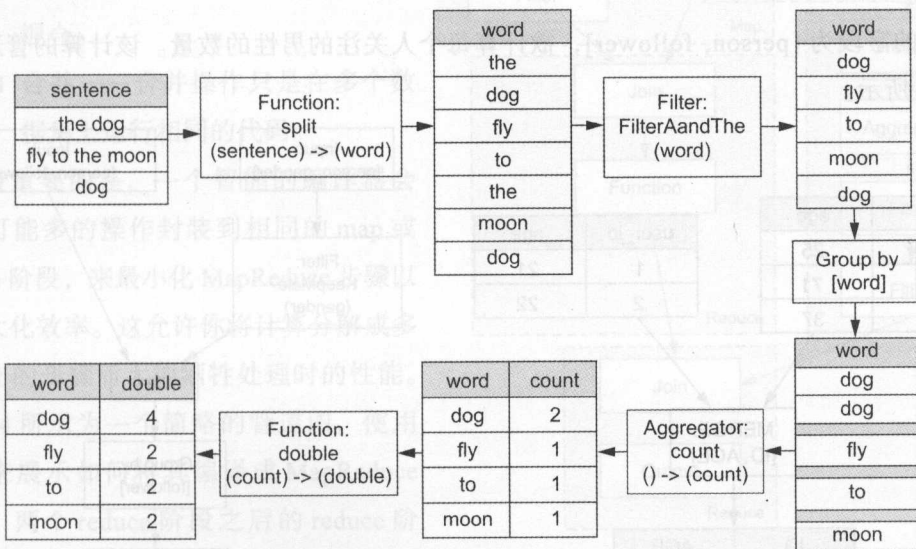


图 6-20 当字段不再被需要时，它将被自动丢弃

第二类操作是合并操作，它将多个独立的元组集合并到单个元组集中。合并操作要求所有元组集具有相同数量的字段，并为元组指定新名称。图 6-22 所示为一个合并示例。

现在来看一个更有趣的例子。假设一个数据集的字段为 [person, gender]，另一个

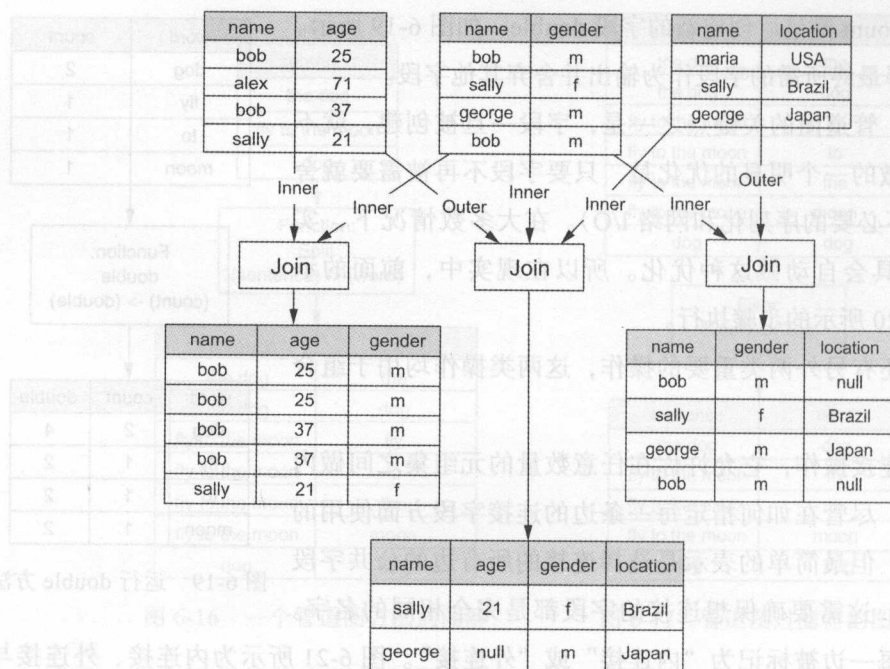


图 6-21 内连接、外连接与混合连接的示例

数据集的字段为 [person, follower], 欲计算每个人关注的男性的数量。该计算的管道图如图 6-23 所示。

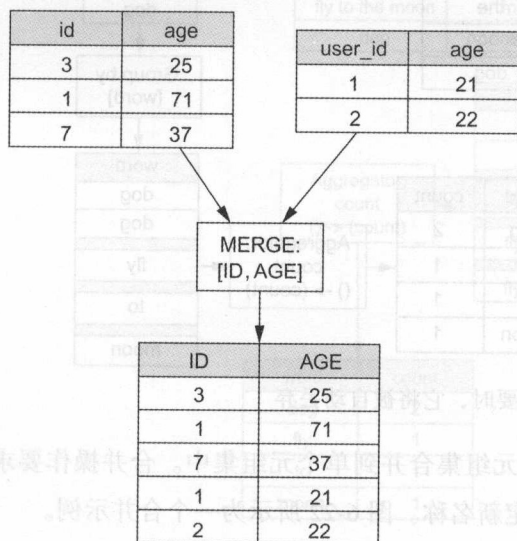


图 6-22 管道图合并操作的示例

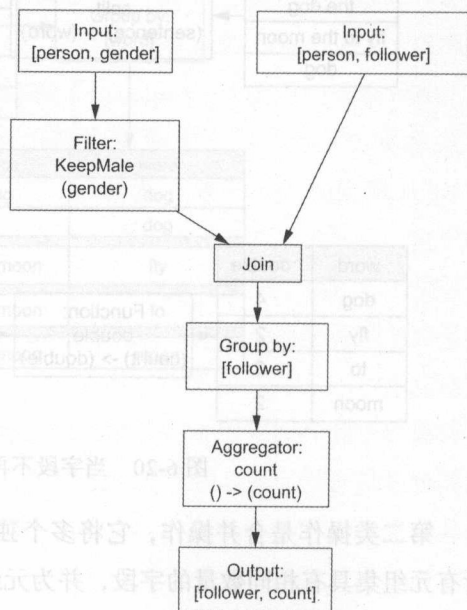


图 6-23 管道图

6.7.2 通过 MapReduce 执行管道图

虽然管道图是一种关于批量计算的高级思维方式，但它们可以被直接编译为一系列 MapReduce 作业，这意味着它们能够以可扩展的方式来执行。每一个管道图操作都可以转换为 MapReduce：

- ❑ **方法和过滤器**——方法和过滤器一次查看一条记录，所以它们可以在连接或聚合之后的 map 阶段或 reduce 阶段中运行。
- ❑ **分组**——通过 map 阶段生成的键，分组很容易被转换成 MapReduce。如果按多个值分组，那么键将是这些值的列表。
- ❑ **聚合**——聚合发生在 reduce 阶段，因为它查看一个组的所有元组。
- ❑ **连接**——你已经知道了实现连接的基本知识，并且知道它们在 map 阶段和 reduce 阶段都需要一些代码。6.6.2 节中双边内连接的代码可以扩展来处理任意数量的边，以及内连接与外连接的任意混合。
- ❑ **合并**——合并操作只是在多个数据集上运行相同的代码。

最重要的是，一个智能的编译器会将尽可能多的操作封装到相同的 map 或 reduce 阶段，来最小化 MapReduce 步骤以及最大化效率。这允许你将计算分解成多个独立的步骤而无须牺牲处理时的性能。图 6-24 所示为一个简略的管道图，使用方框来展示如何将其编译成 MapReduce 作业。两个 reduce 阶段之后的 reduce 阶段意味着在它们之间有一个用于建立连接的 map 阶段。

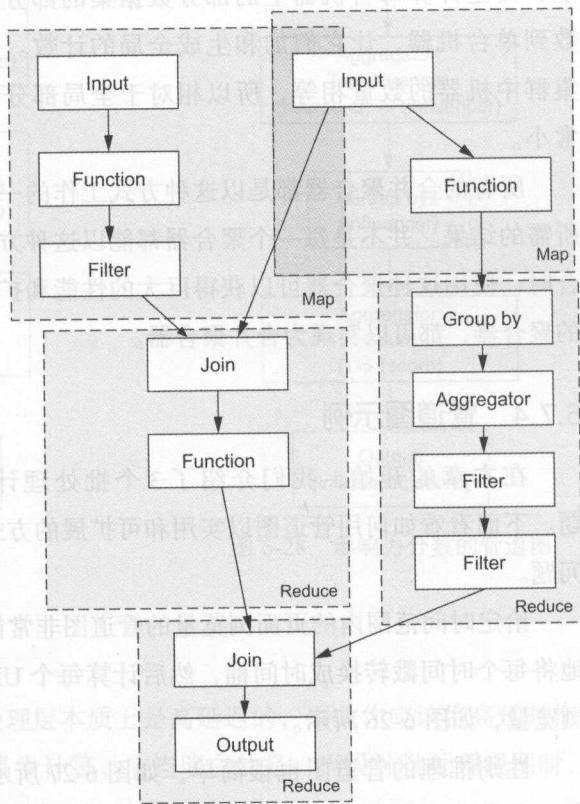


图 6-24 编译成 MapReduce 作业的管道图

6.7.3 合并聚合器

有一种特殊的聚合器比正常聚合器执行得更有效率——合并聚合器。在一些情况下，

使用合并聚合器对扩展性至关重要，且这些情况经常出现，因此了解这些聚合器的工作原理十分重要。

例如，假设要计算数据集中所有记录的计数。管道图如图 6-25 所示。

GroupBy GLOBAL 阶段表明每一个元组应该进入相同的组，并且聚合器应该运行在数据集中的每一个元组上。通常执行的方式是，让每个元组进入同一台机器，然后在这台机器上运行聚合器代码。这是不可扩展的，因为失去了任何可用于并行性的特征。

然而 count 可以更有效地执行——不是发送每个元组到单台机器上，而是计算每台机器上的部分数据集的部分计数，然后发送部分计数到单台机器，让它们加和生成全局的计数。因为部分计数的数量与集群中机器的数量相等，所以相对于全局部分的计算，这个工作量非常小。

所有的合并聚合器都是以这种方式工作的——先做部分聚合，然后合并部分结果得到所需的结果。并不是每一个聚合器都能以这种方式表示，但是当全局聚合或用较少的组聚合时，使用这种聚合器可以获得巨大的性能和扩展性的提升。计数与加和，这两类最常见的聚合器，都可以实现为合并聚合器。

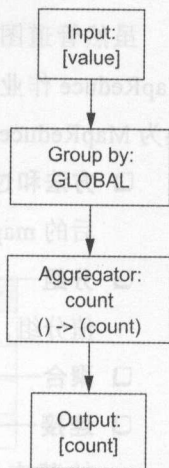


图 6-25 全局聚合

6.7.4 管道图示例

在本章的开始，我们介绍了 3 个批处理计算的示例问题。下面看看如何用管道图以实用和可扩展的方式来解决这些问题。

给定时间范围内的页面浏览量的管道图非常简单，即简单地将每个时间戳转换成时间桶，然后计算每个 URL/桶的页面浏览量，如图 6-26 所示。

性别推理的管道图也很简单，如图 6-27 所示。先简单地归一化每个名字，再使用 maleProbabilityOfName 方法来获得每个名字是男性名字的概率，然后计算人均是男性名字的概率，最后运行一个方法（平均概率大于 0.5 的为男性，低于 0.5 的为女性）对人群进行分类。

最后来看影响力分数的问题。该示例的管道图如图 6-28

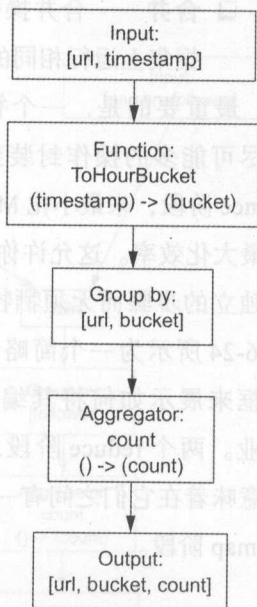


图 6-26 给定时间范围内的页面浏览量的管道图

所示。先通过 `responder-id` 对每个人分组并选择每个人回应最多的影响者，以确定首要影响者，然后只要计算每一个影响者作为其他人的首要影响者出现的次数即可。

正如你所看到的，这些示例问题都能很好地分解成管道图，并且管道图很好地映射了你对数据转换的思考。当在第8章中构建 SuperWebAnalytics.com 的批处理层时，我们需要用到更复杂的计算，你会看到，通过使用这种更高层次的抽象，会节省很多时间和精力。

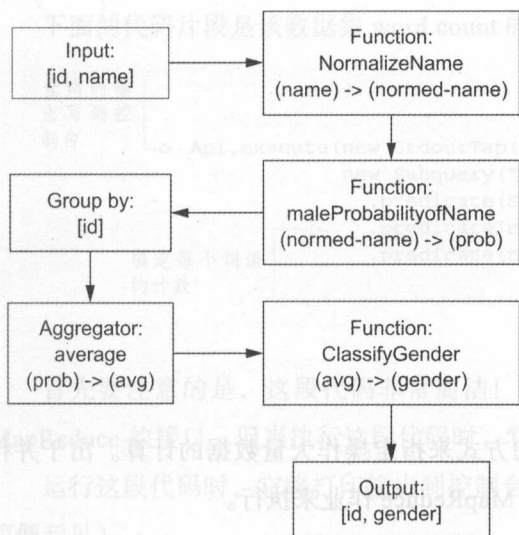


图 6-27 性别推理的管道图

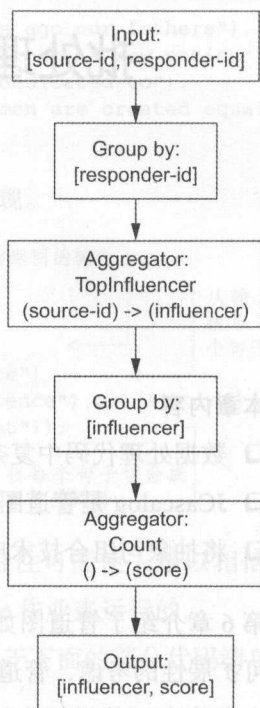


图 6-28 影响力分数的管道图

6.8 总结

批处理层是 Lambda 架构的核心。批处理层本质上是高延迟的，因此你应该将高延时作为契机，完成深入分析和实时中不能做的昂贵计算。应当明白的是，当设计批处理视图时，在生成视图的大小和完成查询所需的工作量之间有一个取舍。

MapReduce 范式以可扩展的方式提供了在所有数据上预先计算查询方法的通用原语，但是很难在 MapReduce 中考虑它。虽然 MapReduce 提供了容错、并行和任务调度，但使用原生 MapReduce 很明显是乏味的，且作用有限。而借助管道图这种想法更简洁、更自然。第7章将介绍名为“JCascalog”的更高层次的抽象，它可以用于实现管道图。

批处理层：示例

本章内容

- ❑ 数据处理代码中复杂性的来源
- ❑ JCascalog 是管道图的现实实现
- ❑ 将抽象与组合技术应用于数据处理

第 6 章介绍了管道图如何以自然、简明的方式来指定操作大量数据的计算。出于并行性和可扩展性的考虑，管道图可以作为一系列 MapReduce 作业来执行。

本章将介绍管道图直接映射的一个工具——JCascalog。在 JCascalog 中有很多要讨论的东西，因此本章比前面的示例章节更复杂。同样，即便不阅读示例章节，你也可以学习 Lambda 架构的完整理论。但是对于 JCascalog，我们的目标是拓宽思路，使你知道对于数据处理工具什么是可能的。一个关键点是，数据处理代码和你编写的其他任何代码没有什么不同——它同样需要满足良好的可重用和可组合的抽象。抽象和组合是好的软件设计的基础。

你不仅要关注 JCascalog 如何实现管道图，我们将超出这个范围，展示 JCascalog 如何实现大量的抽象和组合技术，而这些技术是其他工具不可能有的。大多数开发人员将 SQL 视作数据操作工具的黄金法则，但这种心态有严重的局限性。许多数据处理工具会遇到偶然出现的复杂性，不是因为问题的特性，而是缘于工具本身的设计。本章将讨论其中的一些复杂性，然后展示 JCascalog 是如何避开这些经典陷阱的。你会发现 JCascalog 使用的编程技术让代码编写得非常简洁、非常优雅。

7.1 一个例证

Word count 是典型的 MapReduce 示例，下面来看如何使用 JCascalog 来实现它。

出于引导性目的，我们将显式地输入数据集——Gettysburg 演讲——存储在一个内存列表中，且每个短句被分别存储：

```
List SENTENCE = Arrays.asList(
    Arrays.asList("Four score and seven years ago our fathers"),
    Arrays.asList("brought forth on this continent a new nation"),
    Arrays.asList("conceived in Liberty and dedicated to"),
    Arrays.asList("the proposition that all men are created equal"),
    ...
);
```

为简便起见、省略后边的数据集

下面的代码片段是该数据集 word count 的完整 JCascalog 实现。

```
Api.execute(new StdoutTap(),
    new Subquery("?word", "?count")
        .predicate(SENTENCE, "?sentence")
        .predicate(new Split(), "?sentence").out("?word")
        .predicate(new Count(), "?count"));
```

查询的输出写到控制台

指定查询返回的输出类型

从输入中读取每一个句子

确定每个词语的计数

将每个句子切分成单独的词语

首先要注意的是，这段代码非常简洁！JCascalog 的高级特性可能使人难以相信它是 MapReduce 的接口，但当执行这段代码时，它是作为 MapReduce 作业来运行的。

运行这段代码时，它将打印输出到控制台，返回的结果类似于下面的部分代码清单（为简便起见）：

RESULTS

```
-----
But      1
Four     1
God       1
It        3
Liberty  1
Now       1
The       2
We        2
```

下面通过逐行查看 word-count 代码来理解其功能。即使还没有完全清楚每一个细节，也不必担心，我们将在之后的章节中进一步讲解 JCascalog 的有关内容。

在 JCascalog 中，输入和输出通过一个被称为 **tap** 的抽象来定义。tap 抽象允许将结果显示在控制台、存储在 HDFS 或写入数据库。第一行代码的功能是“执行下列计算并将结

果输出到控制台”：

```
Api.execute(new StdoutTap(), ...
```

第二行代码开始进行计算的定义。计算通过 Subquery 类的实例表示。该子查询会生成一系列包含两个名为“?word”和“?count”字段的元组：

```
new Subquery("?word", "?count")
```

第三行代码为该查询寻求输入数据的来源。它读取 SENTENCE 数据集并生成包含一个名为“?sentence”字段的元组。与输出一样，tap 抽象允许输入来自不同的源，比如内存值、HDFS 文件或其他查询的结果：

```
.predicate(SENTENCE, "?sentence")
```

第四行代码将每个句子切分成一组单词，将“?sentence”字段作为 Split 函数的输入，并将输出存储在一个名为“?word”的新字段中：

```
.predicate(new Split(), "?sentence").out("?word")
```

Split 函数不是 JCascalog API 的一部分，但是它展示了用户新定义的函数如何可以集成到查询中。其操作是通过下面的类来定义的。它的定义是相当直观的——它将句子作为输入，并为句子中的每个单词生成一个新的元组：

```
public static class Split extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        String sentence = call.getArguments().getString(0);
        for (String word: sentence.split(" ")) {
            call.getOutputCollector().add(new Tuple(word));
        }
    }
}
```

将句子切分为单词

为每个单词生成它自己的元组

最后一行代码对每个单词出现的次数进行计数，并将结果存储在“?count”变量中：

```
.predicate(new Count(), "?count"));
```

至此，你已经初步接触了 JCascalog。下面来看一些常见的数据处理工具可以导致不必要复杂性的陷阱。

7.2 数据处理工具的常见陷阱

与任何代码一样，保持数据处理代码简单是至关重要的，这样你就可以推断系统并确保正确性。代码的复杂性以两种形式出现：问题本身固有的、待解决的基本复杂性；仅由

解决方案的方法引起的偶发复杂性。通过最小化偶发复杂性，代码会更容易维护，并可提高其正确性。

本节将介绍数据处理代码中偶发复杂性的两个来源：自定义语言和不良的可组合抽象。

7.2.1 自定义语言

数据处理工具中复杂性的常见来源是自定义语言的使用。这样的例子包括关系数据库的 SQL，或 Hadoop 的 Pig 和 Hive。虽然为数据处理使用自定义语言很吸引人，但却引入了很多严重的复杂性问题。

自定义语言的使用引入了语言障碍——它需要一个接口与其他部分的代码进行交互。该接口是错误的一种常见来源和复杂性不可避免的一种来源。例如，SQL 注入会利用面向用户的代码和为查询关系数据库生成的 SQL 语句之间定义的不恰当的接口产生攻击。因为该接口，你必须时时保持谨慎，以确保自己不出错。

语言障碍也会造成各种其他的复杂性问题。模块化会变得非常令人痛苦——自定义语言可能支持命名空间和函数，但终究不会和它们通用语言对应的部分一样良好。此外，如果想把自己的业务逻辑嵌入查询中，你必须创建用户自定义函数（User Defined Functions, UDFs）并用该语言注册这些函数。

最后，你必须在通用语言和数据处理语言之间协调切换。例如，你可能使用自定义语言编写了一个查询，然后想使用第 5 章的 Pail 类将结果数据附加到现有的存储中。Pail 调用只是标准的 Java 代码，所以你需要编写以正确顺序执行任务的 shell 脚本。因为你工作在通过脚本连接在一起的多种语言上，如果像异常和异常处理等机制发生故障——那么你必须检查返回的代码，以确保之前的步骤失败时下一步骤不会继续执行。

这些都是偶发复杂性的例子，但当数据处理工具是通用语言的一个库时，这些是完全可以避免的，之后你可以将普通代码与数据处理代码自由混合，使用正常的模块化机制，并且异常也会正常工作。正如你将看到的，一个普通的库是简洁的，并能“愉快”地使用自定义语言工作，这都是可能的。

7.2.2 不良的可组合抽象

另一类偶发复杂性的常见来源可能发生在多个抽象相结合的时候。抽象可以组合在一起，以创建新的和更大的抽象，这是很重要的——否则无法重用代码，一旦方法稍有变化，你就得重新编写代码。

一个很好的例子是 Apache Pig（MapReduce 的另一个抽象实现）中的 Average 聚合器。

在撰写本书时，该实现有 300 多行代码和 15 个独立的方法定义。其复杂性源于用来提升协调 map 和 reduce 阶段工作的性能的代码优化。

Pig 的实现的问题是，它重新实现了 Count 和 Sum 聚合器的功能，而没有重用编写这些聚合器的代码。这是不幸的，因为它要维护更多的代码，每次改进 Count 和 Sum 时，这些改变也都需要合并到 Average 中。将 Average 定义为计数聚合器、求和聚合器和除法函数的组合会更好一些。

不幸的是，Pig 的抽象不允许用上述方式来定义 Average。不过，在 JCascalog 中，Average 正是用上述方式定义的：

```
PredicateMacroTemplate Average =
    PredicateMacroTemplate.build("?val")
        .out("?avg")
        .predicate(new Count(), "?count")
        .predicate(new Sum(), "?val").out("?sum")
        .predicate(new Div(), "?sum", "?count").out("?avg");
```

除了它的简单性，Average 的定义和 Pig 实现一样高效，因为它重用了之前优化的 Count 和 Sum 聚合器。之所以 JCascalog 允许这种组合而 Pig 不允许，完全是因为 JCascalog 和 Pig 中表达计算的差异。稍后将详细讨论 JCascalog 的这个功能——这里的关键是“抽象是可组合的重要性”。本章还将探索许多组合的其他例子。

至此，你已经了解了数据处理工具中复杂性的一些常见来源，现在开始讨论 JCascalog。

7.3 JCascalog 介绍

JCascalog 是一个 Java 库，它为表达 MapReduce 计算提供了可组合的抽象。回想一下，本书旨在说明大数据的概念，并使用具体的工具落实这些概念。还有其他为 MapReduce 提供高级接口的工具——Hive、Pig 和 Cascading 是最流行的——但是许多工具在抽象和组合数据处理代码的能力上仍有局限性。之所以选择 JCascalog，因为它是专门为降低批处理的复杂性而启用新的抽象和组合技术来编写的。

JCascalog 是一种声明性的抽象，其中的计算通过逻辑约束来表示。它不是为如何获得所需的输出提供显式说明，而是根据输入来描述输出。从这个描述中，JCascalog 会确定最高效的方式并通过一系列 MapReduce 作业来执行计算。

如果接触过关系型数据库，那么你会觉得 JCascalog 既陌生又熟悉的——尽管它们在不同的包装中，你仍会识别出一些熟悉的概念，如声明性编程、连接和聚合。但它可能似乎又是不同的，因为它不是 SQL，而是基于逻辑编程的 API。

7.3.1 JCascalog 的数据模型

JCascalog 的数据模型和第6章中管道图的数据模型是一样的。JCascalog 操作和转换的元组——值的命名列表，每个值可以是任何类型的对象。一组元组共享用以指定每个元组有多少字段和每个字段名称的模式。图7-1所示为共享模式的一组元组的示例。

?name	?age	?gender
"alice"	28	"f"
"jim"	48	"m"
"emily"	21	"f"
"david"	25	"m"

- ① 该共享模式定义了元组包含的每个字段的名称。
- ② 每个元组对应于一条独立的记录，并且可以包含不同类型的数据。

当执行一个查询时，JCascalog 将初始数据表示为元组，并在计算的每个阶段中将输入转换成一系列的其他元组集。

图 7-1 附带模式描述的元组示例集合

大量的标点符号 ?!

在看到 JCascalog 的例子后，一个自然的问题是，“所有这些问号是什么意思？”很高兴你能这么问。

名称以问号 (?) 开始的字段是非空的。对于非空字段，如果 JCascalog 遇到一个带有空值的元组，它会立即从工作数据集中将其过滤掉；相反，名称以感叹号 (!) 开始的字段可以包含空值。

此外，名称以双感叹号 (!!) 开始的字段也可以为空值，并需要在数据集之间执行外连接。因为连接涉及这些类型的字段名，不满足数据集之间连接条件的记录仍然包含在结果集中，但是这些数据不存在的字段为空值。

通过各种例子来介绍 JCascalog 是最好的方式。除了之前提到的 SENTENCE 数据集，下面还将使用其他一些内存中的数据集来演示 JCascalog 的不同方面。这些数据集示例如图7-2所示，本书附带的源代码包中的完整数据集是可用的。

AGE		GENDER		FOLLOWS		INTEGER
?person	?age	?person	?gender	?person	?follows	?num
"alice"	28	"alice"	"f"	"alice"	"david"	-1
"bob"	33	"bob"	"m"	"alice"	"bob"	0
"chris"	40	"chris"	"m"	"bob"	"david"	1
"david"	25	"emily"	"f"	"emily"	"gary"	2

图 7-2 用以演示 JCascalog API 的示例数据集：人的年龄的数据集、单独的性别的数据集、person-following 关系的数据集（像 Twitter 中的那样）和整数的数据集

JCascalog 受益于能够表达复杂查询的简单语法。下面将检查 JCascalog 的查询结构。

7.3.2 JCascalog 查询的结构

JCascalog 查询有统一的结构，它由目的 tap 和定义了实际计算的子查询组成。试思考下面的例子，找出 AGE 数据集中 30 岁以下的所有人：

```
Api.execute(new StdoutTap(),
            new Subquery("?person")
                .predicate(AGE, "?person", "?age")
                .predicate(new LT(), "?age", 30));
```

目的 tap。 输出字段。 定义所需输出的谓词。

注意：JCascalog 不是表达如何执行计算，而是使用谓词来描述所需的输出。这些谓词能够表达元组集上所有可能的操作——转换、过滤、连接等——它们可以被分为 4 种主要类型：

- ❑ **函数谓词**指定一组输入字段和一组输出字段之间的关系。加法、乘法等数学函数都属于这一类，但函数也可以由单个输入元组生成多个元组。
- ❑ **过滤器谓词**指定一组输入字段的约束，并移除所有不满足约束的元组。小于和大于操作都是这种类型的例子。
- ❑ **聚合器谓词**是针对一组元组的函数。例如，聚合器可以计算平均数，为整个组生成单个输出。
- ❑ **生成器谓词**只是元组的有限集合。生成器可以是具体的数据源（例如内存中的数据或 HDFS 文件），也可以是另一个子查询的结果。

附加的示例谓词如图 7-3 所示。

类型	示例	描述
生成器	<code>.predicate(SENTENCE, "?sentence")</code>	从 SENTENCE 数据集创建元组的生成器，每个元组包含一个名为“?sentence”的字段
函数	<code>.predicate(new Multiply(), 2, "?x").out("?z")</code>	该函数加倍“?x”的值并将结果存储为“?z”
过滤器	<code>.predicate(new LT(), "?y", 50)</code>	过滤器移除“?y”的值大于等于 50 的所有元组

图 7-3 生成器、函数和过滤器谓词的示例。我们将在本章的稍后部分讨论聚合器，但它们共享相同的结构

JCascalog 的关键设计决策是使所有谓词共享一个共同的结构。谓词的第一个参数是谓

词的操作，其余参数都是该操作的参数。对于函数和聚合器谓词，使用 `out` 方法指定输出的标签。

能够通过同样简单一致的机制来表达每一步计算，是使抽象高度可组合的关键。尽管谓词的结构简单，但它们提供了极其丰富的语义。示例如图 7-4 所示。

类型	示例	描述
函数作为过滤器	<code>.predicate(new Plus(), 2, "?x").out(6)</code>	尽管 <code>Plus()</code> 是一个函数，但这个谓词过滤了所有“? <code>x</code> ”的值不等于 4 的元组。
复合过滤器	<code>.predicate(new Multiply(), 2, "?a").out("?z")</code> <code>.predicate(new Multiply(), 3, "?b").out("?z")</code>	这两个谓词结合起来可以过滤“ <code>2(?a) ≠ 3(?b)</code> ”的元组。

图 7-4 简单的谓词结构可以表达深层语义关系，来描述所需的查询输出

正如之前提到的，数据集之间的连接也通过谓词来表示——下面将扩展这方面的内容。

7.3.3 查询多个数据集

许多查询会要求结合多个数据集，在关系型数据库中，最常见的是通过连接操作来完成，JCascalog 中也存在连接。

假设你想通过合并 AGE 和 GENDER 数据集，来创建包含存在于两个数据集中所有人年龄和性别的元组的新集合。这是“?`person`”字段上标准的内连接，如图 7-5 所示。

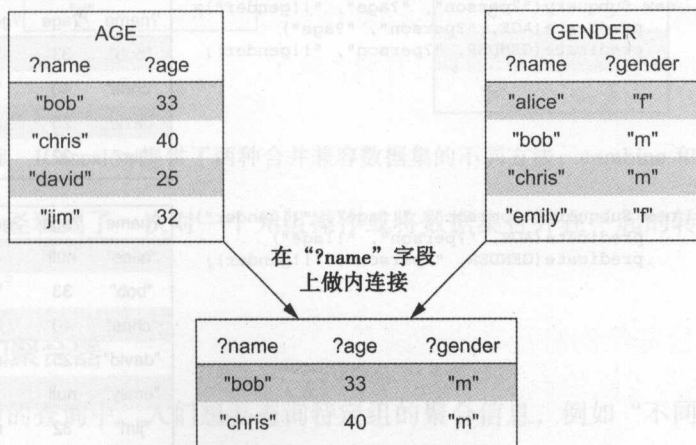


图 7-5 AGE 和 GENDER 数据集的内连接合并了存在于两个数据集的“?`person`”值的元组数据

在类 SQL 的语言中，连接是显式表达的；JCascalog 中基于变量名的连接是隐式的。

图 7-6 所示为它们之间的差异。

语 言	查 询	描 述
SQL	<pre>SELECT AGE.person, AGE.age, GENDER.gender FROM AGE INNER JOIN GENDER ON AGE.person = GENDER.person]</pre>	这个子句显式地定义了连接条件。
JCascalog	<pre>new Subquery("?person", "?age", "?gender") .predicate(AGE, "?person", "?age") .predicate(GENDER, "?person", "?gender");</pre>	通过指定“?person”作为这两个数据集的字段名，JCascalog使用该共享名称进行隐式连接。

图 7-6 SQL 和 JCascalog 对于 AGE 和 GENDER 数据集之间内连接的语法比较

JCascalog 中连接的工作方式与管道图中连接的工作方式相同：通过以公共字段名作为连接键，将元组集连接在一起。在该查询中，相同字段名“?person”作为两种不同生成器谓词（AGE 和 GENDER）的输出。因为变量的每个实例对于任何产生的元组都必须具有相同的值，所以 JCascalog 知道解决该查询的正确方式是在 AGE 和 GENDER 数据集之间做一个内连接。

内连接只生成连接字段存在于连接所有边的元组。但在有些情况下，你可能希望得到不存在于某个数据集的记录的结果，从而导致不存在的数据为空值。这些操作被称为外连接，在 JCascalog 中完成同样简单。示例如图 7-7 所示。

连接类型	查 询	结 果																					
左外连接	<pre>new Subquery("?person", "?age", "!!gender") .predicate(AGE, "?person", "?age") .predicate(GENDER, "?person", "!!gender");</pre>	<table> <thead> <tr> <th>?name</th><th>?age</th><th>?gender</th></tr> </thead> <tbody> <tr> <td>"bob"</td><td>33</td><td>"m"</td></tr> <tr> <td>"chris"</td><td>40</td><td>"m"</td></tr> <tr> <td>"david"</td><td>25</td><td>null</td></tr> <tr> <td>"jim"</td><td>32</td><td>null</td></tr> </tbody> </table>	?name	?age	?gender	"bob"	33	"m"	"chris"	40	"m"	"david"	25	null	"jim"	32	null						
?name	?age	?gender																					
"bob"	33	"m"																					
"chris"	40	"m"																					
"david"	25	null																					
"jim"	32	null																					
全外连接	<pre>new Subquery("?person", "!!age", "!!gender") .predicate(AGE, "?person", "!!age") .predicate(GENDER, "?person", "!!gender");</pre>	<table> <thead> <tr> <th>?name</th><th>?age</th><th>?gender</th></tr> </thead> <tbody> <tr> <td>"alice"</td><td>null</td><td>"f"</td></tr> <tr> <td>"bob"</td><td>33</td><td>"m"</td></tr> <tr> <td>"chris"</td><td>40</td><td>"m"</td></tr> <tr> <td>"david"</td><td>25</td><td>null</td></tr> <tr> <td>"emily"</td><td>null</td><td>"f"</td></tr> <tr> <td>"jim"</td><td>32</td><td>null</td></tr> </tbody> </table>	?name	?age	?gender	"alice"	null	"f"	"bob"	33	"m"	"chris"	40	"m"	"david"	25	null	"emily"	null	"f"	"jim"	32	null
?name	?age	?gender																					
"alice"	null	"f"																					
"bob"	33	"m"																					
"chris"	40	"m"																					
"david"	25	null																					
"emily"	null	"f"																					
"jim"	32	null																					

图 7-7 在 AGE 和 GENDER 数据集之间，JCascalog 查询实现两种类型的外连接

正如前面提到的，对于外连接，JCascalog 使用以“!!”开始的字段，为不存在的数据

生成空值。在左外连接中，结果集中一个人必须有一个年龄，对于缺失的性别数据可以引入空值。在全外连接中，两个数据集中存在的所有人都要包含在结果中，对任何缺失的年龄或性别数据使用空值。

除了连接，还有一些可以合并数据集的方法。有时，你有两个包含相同类型数据的数据集，并且想要将它们合并成单个的数据集。为此，JCascalog 提供了 `combine` 和 `union` 方法：`combine` 方法将数据集简单合并在一起；而 `union` 方法将在合并过程中删除任何重复的记录。两种方法之间的区别如图 7-8 所示。

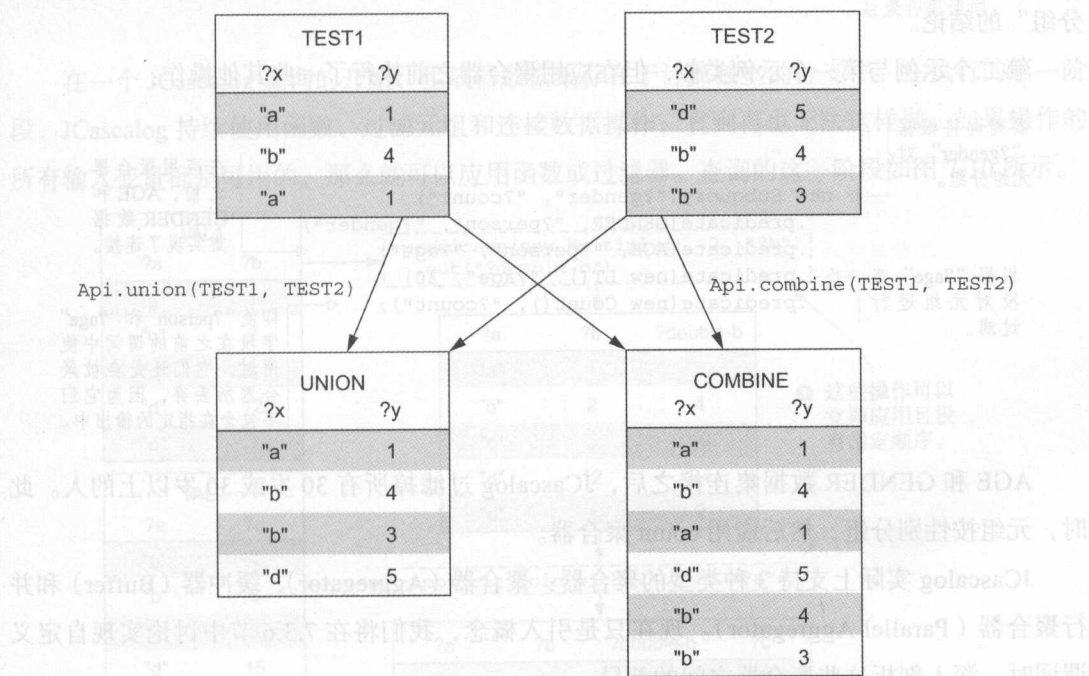


图 7-8 JCascalog 提供了两种合并兼容数据集的不同方法：`combine` 和 `union`

至此，你已经看到了一次对一个元组操作或将数据集合并在一起的转换。下面讨论处理元组的组操作。

7.3.4 分组和聚合器

在许多类型的查询中，人们想要查询特定组的聚合信息，例如“不同职业的平均工资是多少？”或“什么年龄段写微博最多？”。在 SQL 中，如何对记录分组以及如何执行结果集上的操作是被显式声明的。

JCascalog 中没有明确的 `GROUP BY` 命令用以指示如何为了聚合对元组分区。相反，

与连接一样，基于所需查询输出的分组是隐式的。为了说明这一点，下面来看几个示例。

第一个示例使用 Count 聚合器得到每个人 follow 的人数：

下划线提示 JCascalog 忽略该字段。

```
new Subquery("?person", "?count")
    .predicate(FOLLOWS, "?person", "_")
    .predicate(new Count(), "?count");
```

输出字段名称定义了所有可能的分组。

当执行该聚合器时，输出字段意味着元组应该按照“?person”分组。

当 JCascalog 执行 count 谓词时，它从声明的输出中得到“必须首先完成‘?person’上分组”的结论。

第二个示例与第一个示例类似，但在应用聚合器之前执行了一些其他操作：

该查询将根据“?gender”对元组分组。

```
new Subquery("?gender", "?count")
    .predicate(GENDER, "?person", "?gender")
    .predicate(AGE, "?person", "?age")
    .predicate(new LT(), "?age", 30)
    .predicate(new Count(), "?count");
```

根据“?age”字段对元组进行过滤。

在应用聚合器之前，AGE 和 GENDER 数据集实现了连接。

即使“?person”和“?age”字段在之前的谓词中使用过，它们还是会被聚合器所丢弃，因为它们不包含在指定的输出中。

AGE 和 GENDER 数据集连接之后，JCascalog 过滤掉所有 30 岁或 30 岁以上的人。此时，元组按性别分组，然后应用 count 聚合器。

JCascalog 实际上支持 3 种类型的聚合器：聚合器（Aggregator）、缓冲器（Buffer）和并行聚合器（Parallel Aggregator）。现在只是引入概念，我们将在 7.3.6 节中讨论实现自定义谓词时，深入剖析这些聚合器之间的差异。

我们已经详细地讨论了 JCascalog 不同类型谓词之间的差异。下面对一个查询示例进行单步调试，来看看在查询计算的不同阶段元组集是如何被操作的。

7.3.5 对一个查询示例进行单步调试

对于这个例子，我们将从两个测试数据集开始，如图 7-9 所示。

我们将使用以下查询来阐释 JCascalog 查询的执行，观察执行的每个阶段元组集是如何

VAL1		VAL2	
?a	?b	?a	?c
"a"	1	"b"	4
"b"	2	"b"	6
"c"	5	"c"	3
"d"	12	"d"	15
"d"	1		

图 7-9 查询用的测试数据

变化的：

```

测试数
据集的
生成器。
new Subquery("?a", "?avg")
.predicate(VAL1, "?a", "?b")
.predicate(VAL2, "?a", "?c")
.predicate(new Multiply(), 2, "?b").out("?double-b")
.predicate(new LT(), "?b", "?c")
.predicate(new Count(), "?count")
.predicate(new Sum(), "?double-b").out("?sum")
.predicate(new Div(), "?sum", "?count").out("?avg")
.predicate(new Multiply(), 2, "?avg").out("?double-avg")
.predicate(new LT(), "?double-avg", 50);

多个聚
合器。

预先聚合
器函数和
过滤器。

后聚合器谓词。

```

在一个 JCascalog 查询的开始，生成器数据集存在于计算独立的分支中。在执行的第一阶段，JCascalog 持续使用函数、过滤元组和连接数据操作，直到再也无法这样做。如果操作的所有输入变量都是可用的，那么就可以应用函数或过滤器。查询的这一阶段如图 7-10 所示。

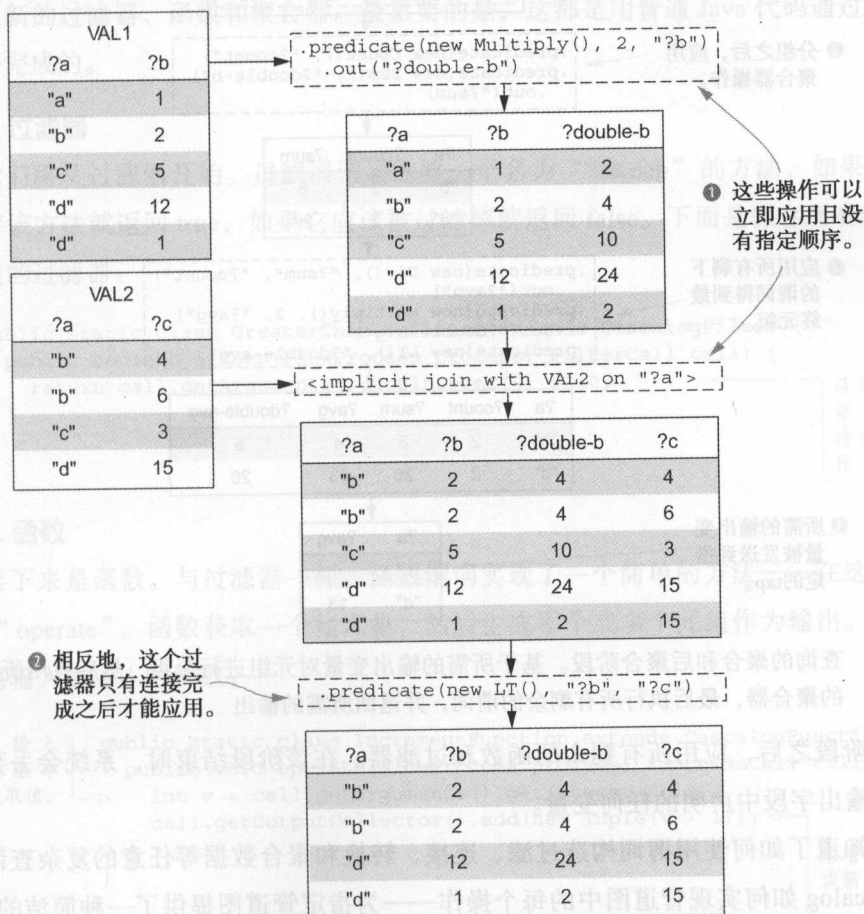


图 7-10 当输入变量可用时，执行的第一阶段需要应用所有函数、过滤器和连接

注意：一些谓词需要先应用其他谓词。在这个示例中，直到连接操作执行完之后，小于过滤器才能应用。

最终该阶段达到没有谓词可以应用的情况，因为剩下的谓词要么是聚合器，要么需要的变量还不可用。此时，JCascalog 将进入查询的聚合阶段。

JCascalog 基于查询中声明为输出变量的任何可用变量对元组进行分组，然后将聚合器应用到元组的每个组中，如图 7-11 所示。

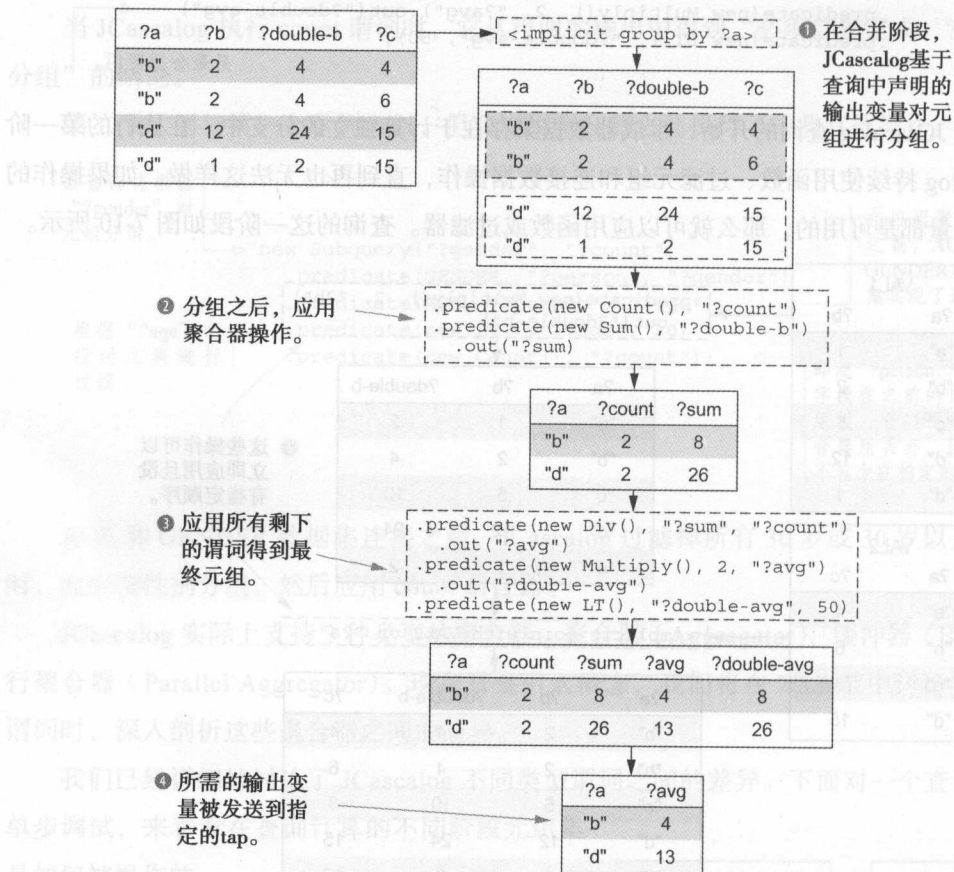


图 7-11 查询的聚合和后聚合阶段。基于所需的输出变量对元组进行分组，然后应用所有的聚合器，最后执行所有剩余的谓词，并返回所需的输出

在聚合阶段之后，应用所有剩余的函数和过滤器。在该阶段结束时，系统会丢弃元组中未在查询输出字段中声明的任何变量。

你已经知道了如何使用谓词构造过滤、连接、转换和聚合数据等任意的复杂查询，也知道了 JCascalog 如何实现管道图中的每个操作——为指定管道图提供了一种简洁的方式。

下面演示如何实现自己的自定义过滤器、函数和聚合器，来用作 JCascalog 的谓词。

详细的解释

你可能已经注意到，这个示例通过完成计数、求和与除法来计算平均数。这只是为了演示之用——这些操作可以抽象成 Average 聚合器，正如你在本章前面所看到的。

你可能也注意到，经过某个阶段后，一些变量从未被使用过，但仍保留在生成的元组集中。例如，“?b”变量在 LT 谓词应用后就没有被使用过，但它仍然和其他变量一起被分组。事实上，变量一旦不再被需要，JCascalog 就会丢弃它们，因此它们不用被序列化或通过网络传输。这是前一章中提到的可以应用于任何管道图的优化。

7.3.6 自定义谓词操作

你经常需要创建另外的谓词类型来实现业务逻辑。为此，JCascalog 展示了简单的接口来定义新的过滤器、函数和聚合器。最重要的是，这都是用普通 Java 代码通过实现适当的接口来完成的。

1. 过滤器

我们将从过滤器开始。过滤器谓词需要一个名为“isKeep”的方法，如果输入元组应该保存该方法就返回 true，如果它应该被过滤掉就返回 false。下面是保存所有输入大于 10 的元组的过滤器：

```
public static class GreaterThanTenFilter extends CascalogFilter {
    public boolean isKeep(FlowProcess process, FilterCall call) {
        return call.getArguments().getInteger(0) > 10;
    }
}
```

获得输入元组的第一个元素，并将该元素的值看作一个整数。

2. 函数

接下来是函数。与过滤器一样，函数谓词实现了一个简单的方法——在这个例子中命名为“operate”。函数获取一个输入集，然后生成零个或多个元组作为输出。这是一个简单的将输入值加 1 的函数：

```
public static class IncrementFunction extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        int v = call.getArguments().getInteger(0);
        call.getOutputCollector().add(new Tuple(v + 1));
    }
}
```

从输入元组中获取值。

用增量的值生成新的元组。

图 7-12 所示为将这个函数应用到元组集上的结果。

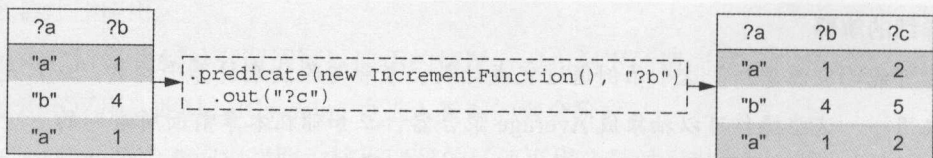


图 7-12 应用到样本元组上的 IncrementFunction 谓词

之前说过，对于给定的元组，如果一个函数生成零个元组，那么该函数可以作为过滤器使用。下面是一个试图将字符串解析成整数的函数，如果解析失败就过滤掉元组：

```

public static class TryParseInteger extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        String s = call.getArguments().getString(0);
        try {
            int i = Integer.parseInt(s);
            call.getOutputCollector().add(new Tuple(i));
        } catch (NumberFormatException e) {}
    }
}

```

将输入值看作字符串。 →

如果解析成功，则生成作为整数值。 ←

如果解析失败，则什么都不生成。 ←

图 7-13 展示了将该函数应用到一个元组集。可以看到，一个元组被该方法过滤掉了。

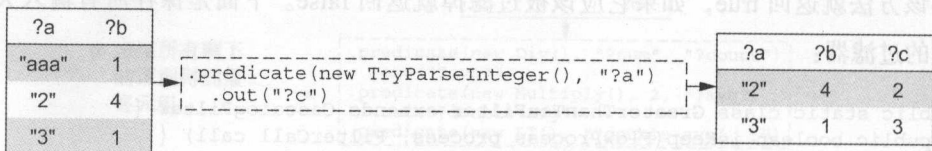


图 7-13 TryParseInteger 函数过滤掉 ?a 不能转换为整数值的那一行

最后，如果一个函数生成多个输出元组，那么每个输出元组被追加到输入参数的副本上。例如，word count 中的 Split 函数如下：

```

public static class Split extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        String sentence = call.getArguments().getString(0);
        for(String word: sentence.split(" ")) {
            call.getOutputCollector().add(new Tuple(word));
        }
    }
}

```

生成每个单词作为独立的元组。 →

为简单起见，使用空格符来分割单词。 ←

图 7-14 所示为将这个函数应用到语句集的结果。可以看到，每个输入句子会复制它包含每个的单词。

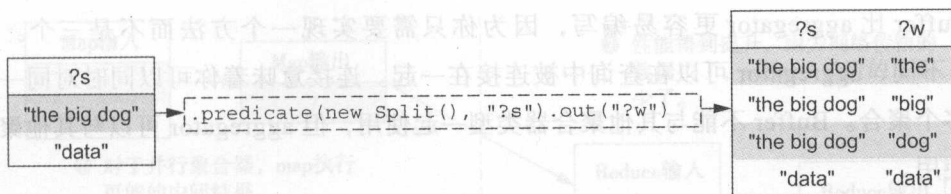


图 7-14 Split 函数可以由单个输入元组生成多个元组

3. 聚合器

最后一类可自定义的谓词操作是聚合器。正如前面所提到的，有 3 种类型的聚合器，每一种在组合与性能方面都有不同的属性。

也许很明显，第一种类型的聚合器从字面上被称为 **aggregator**。对于一个组中的每个元组，**aggregator** 一次只查看一个元组，并为每个所查看的元组调整内部状态。下面是将加和作为 **aggregator** 的一个实现：

```

初始化聚合器内部状态。
public static class SumAggregator extends CascalogAggregator {
    public void start(FlowProcess process, AggregatorCall call) {
        call.setContext(0);
    }

    public void aggregate(FlowProcess process, AggregatorCall call) {
        int total = (Integer) call.getContext();
        call.setContext(total + call.getArguments().getInteger(0));
    }

    public void complete(FlowProcess process, AggregatorCall call) {
        int total = (Integer) call.getContext();
        call.getOutputCollector().add(new Tuple(total));
    }
}

调用每个元组；更新内部状态来存储运行着的加和。
一旦所有元组被处理完，就生成一个带有最终结果的元组。

```

第二种类型的聚合器被称为 **buffer**，**buffer** 接收一个迭代器并传送给一个组中的整个元组集。使用 **buffer** 实现加和的代码如下：

```

public static class SumBuffer extends CascalogBuffer {
    public void operate(FlowProcess process, BufferCall call) {
        Iterator<TupleEntry> it = call.getArgumentsIterator();
        int total = 0;
        while(it.hasNext()) {
            TupleEntry t = it.next();
            total += t.getInteger(0);
        }
        call.getOutputCollector().add(new Tuple(total));
    }
}

通过迭代器获取元组集。
一个函数遍历所有元组并生成输出元组。

```

Buffer 比 aggregator 更容易编写，因为你只需要实现一个方法而不是三个。但与 buffer 不同，aggregator 可以在查询中被连接在一起。连接意味着你可以同时对同一个组计算多个聚合。Buffer 不能与其他聚合器类型一起使用，但 aggregator 可以与其他聚合器一起使用。

在 MapReduce 框架的上下文中，buffer 和 aggregator 都依赖 reducer 来执行这些操作的实际计算，如图 7-15 所示。

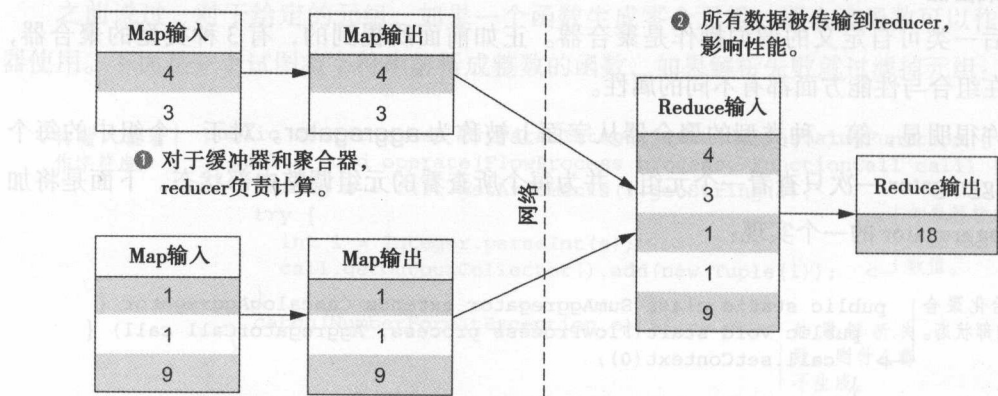


图 7-15 MapReduce 层面的求和 aggregator 与求和 buffer 的执行

JCascalog 将尽可能多的操作封装到 map 和 reduce 任务中，但这些聚合器操作仅由 reducer 执行。这需要一个网络密集型的方法，因为计算的所有数据必须从 mapper 流动到 reducer。此外，如果只有一个组（例如，如果对数据集中的元组数量进行计数），那么为了聚合，所有元组必须被发送到同一个 reducer，使用并行计算系统的目的就会失败。

幸运的是，最后一类聚合器运算可以将聚合操作做得更可扩展、更高效。这些聚合器类似于管道图中的合并聚合器，但是在 JCascalog 中它们被称作并行聚合器。并行聚合器通过在 map 任务中完成部分聚合，来增量地执行聚合。

图 7-16 所示为将求和实现为并行聚合器时的分工。并不是每一个聚合器都可以实现为并行聚合器，但当可能实现时，你就可以通过避免所有网络 I/O 来获得巨大的性能提升。

要编写自己的并行聚合器，你必须实现两个方法：

- ❑ init 函数将单个元组中的参数映射到该元组的部分聚合中。
- ❑ combine 函数指定如何将两部分的聚合值合并为单个聚合值。

下面的代码实现了求和的并行聚合器：

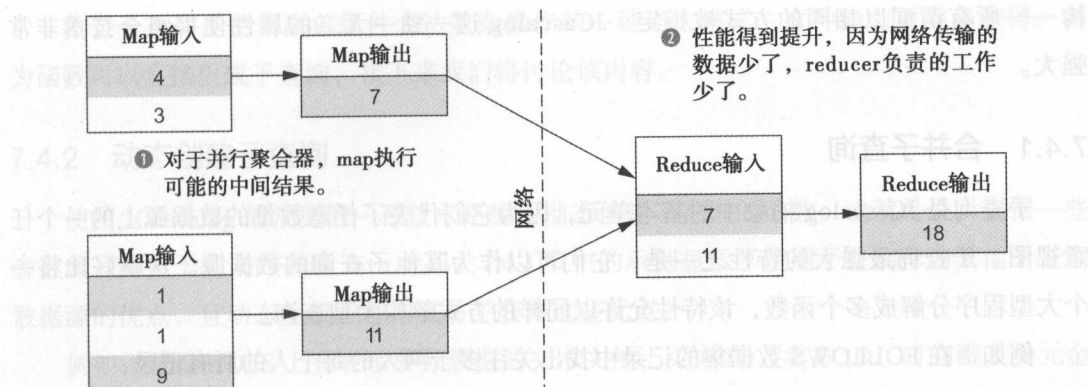


图 7-16 MapReduce 层面的求和并行聚合器的执行

对于求和，
部分聚合的
只是参数中
的值。

```
public static class SumParallel implements ParallelAgg {
    public void prepare(FlowProcess process, OperationCall call) {}

    public List<Object> init(List<Object> input) {
        return input;
    }

    public List<Object> combine(List<Object> input1,
        List<Object> input2) {
        int val1 = (Integer) input1.get(0);
        int val2 = (Integer) input2.get(0);
        return Arrays.asList((Object) (val1 + val2));
    }
}
```

要合并两部分聚合
值，只要将值相加
即可。

并行聚合器可以与其他并行聚合器或普通聚合器连接在一起。但当与普通聚合器连接时，并行聚合器无法在 map 任务中做部分聚合，其功能将与普通聚合器相同。

你现在已经看到了构成 JCascalog 子查询的所有抽象：谓词、函数、过滤器和聚合器。这些抽象的作用在于它们如何促进重用和可组合性。下面来看 JCascalog 各种可能的组合技术。

7.4 组合

在讨论最小化数据处理代码中的偶发复杂性时，我们强调抽象应该用组合来创建新的和更大的功能。这种原理在 JCascalog 中无处不在。

在本节中，我们将通过整合子查询、谓词宏和函数来动态地创建子查询和宏，讨论组成的抽象。这些技术利用“查询工具和通用编程语言之间没有障碍”这一事实，让你以非常细粒度的方式来操纵查询。这些技术还利用了 JCascalog 的令人难以置信的统一结

构——所有谓词以相同的方式被指定。JCascalog 这一独一无二的属性使得组合技术非常强大。

7.4.1 合并子查询

子查询是 JCascalog 抽象中的基本单元，因为它们代表了任意数量的数据源上的一个任意视图。子查询最强大的特性之一是，它们可以作为其他子查询的数据源。这就好比将一个大型程序分解成多个函数，该特性允许以同样的方式解构大型查询。

例如，在 FOLLOWS 数据集的记录中找出关注多于两人的每个人的所有记录：

```

Subquery manyFollows = new Subquery("?person")
    .predicate(FOLLOWS, "?person", "_")
    .predicate(new Count(), "?count")
    .predicate(new GT(), "?count", 2);

Api.execute(new StdoutTap(),
    new Subquery("?person1", "?person2")
        .predicate(manyFollows, "?person1")
        .predicate(manyFollows, "?person2")
        .predicate(FOLLOWS, "?person1", "?person2"));

```

第一个子查询用于确定关注多于两个人的所有人。

只考虑关注者，而不是源。

对每个用户关注的人数进行计数，保存计数值大于2的那些用户。

使用第一个子查询的结果作为这个子查询的源。

只保存关注者和源都存在于第一个子查询结果中的记录。

子查询是延迟执行的——在调用 `Api.execute` 之前，它是什么都不计算的。在前面的示例中，尽管首先定义了 `manyFollows` 子查询，但 `Api.execute` 被调用之前，是没有 MapReduce 作业启动的。

下面是另一个需要多个子查询的查询示例。该查询通过找出存在于每个计算完的单词计数中单词的数量，来扩展 `word count`：

```

Subquery wordCount = new Subquery("?word", "?count")
    .predicate(SENTENCE, "?sentence")
    .predicate(new Split(), "?sentence").out("?word")
    .predicate(new Count(), "?count");

Api.execute(new StdoutTap(),
    new Subquery("?count", "?num-words")
        .predicate(wordCount, "_", "?count")
        .predicate(new Count(), "?num-words"));

```

基本的 word count 子查询。

第二个子查询只需要每个单词的计数。

对于每个计数值确定单词的数量。

整合子查询是使用简单组件表达复杂操作的一种强大模式。这种能力更容易获得，因为函数可以直接生成子查询，接下来我们将讨论该内容。

7.4.2 动态创建子查询

使用 JCascalog 最常见的一种技术是编写动态创建子查询的函数。也就是说，根据一些参数编写构造子查询的普通 Java 代码。7.4.3 节中的示例显示了使用子查询作为其他子查询数据源的优点，且动态生成子查询更易于获得这些优点。

例如，假设 HDFS 上有表示事务数据的文本文件：买方 ID、卖方 ID、时间戳及美元金额。该数据是 JSON 编码的，如下所示：

```
{ "buyer": 123, "seller": 456, "amt": 50, "timestamp": 1322401523 }
{ "buyer": 1009, "seller": 12, "amt": 987, "timestamp": 1341401523 }
{ "buyer": 2, "seller": 98, "amt": 12, "timestamp": 1343401523 }
```

你可能想在这些数据上运行各种各样的计算，但是每一个查询都有从文本文件中解析数据的需求。一个有用的通用函数会将一个 HDFS 路径作为输入，并返回这个位置上解析数据的子查询：

将 JSON 转换为 map 映射的一个内部库。

所需的 map 映射值被转换到单个元组中。

从提供的 HDFS 路径中生成 tap

```
public static class ParseTransactionRecord extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        String line = call.getArguments().getString(0);
        Map parsed = (Map) JSONValue.parse(line);
        call.getOutputCollector().add(new Tuple(parsed.get("buyer"),
            parsed.get("seller"),
            parsed.get("amt"),
            parsed.get("timestamp")));
    }
}
```

该子查询需要用一个 Cascalog 函数来执行实际的解析。

动态创建子查询的普通 Java 函数。

```
public static Subquery parseTransactionData(String path) {
    return new Subquery("?buyer", "?seller", "?amt", "?timestamp")
        .predicate(Api.hfsTextline(path), "?line")
        .predicate(new ParseTransactionRecord(), "?line")
        .out("?buyer", "?seller", "?amt", "?timestamp");
}
```

调用自定义 JSON 解析函数

一旦该抽象被定义，你可以使用这个抽象完成该数据集上的任何查询。例如，计算每个买家交易数量的一个查询，代码如下：

```

public static Subquery buyerNumTransactions(String path) {
    return new Subquery("?buyer", "?count")
        .predicate(parseTransactionData(path), "?buyer", "_", "_", "_")
        .predicate(new Count(), "?count");
}

```

忽略除了买家之外的其他所有字段。

这是一个非常简单的动态创建子查询的示例，但它阐明了如何通过将子查询组合在一起来抽象更复杂的计算。下面通过另一个示例来看子查询中谓词的数量是如何基于参数动态地创建的。

假设有一组转发微博的数据集，每条记录表示对其他微博的转发，并且你想找到固定长度的转发微博的所有计算链。也就是说，对于长度为 4 的计算链，你想知道微博的所有转发。

原始数据集包含一对微博标识符。**注意：**你可以通过连接数据集本身，将微博标识符对转换成长度为 3 的计算链。类似地，你可以通过连接长度为 3 的计算链和原始的微博标识符对，找到长度为 4 的计算链。为了说明这一点，下面给出一个给定输入微博标识符对的生成器，返回计算链长度为 3 的查询：

```

public static Subquery chainsLength3(Object pairs) {
    return new Subquery("?a", "?b", "?c")
        .predicate(pairs, "?a", "?b")
        .predicate(pairs, "?b", "?c");
}

```

另外一个连接找到所有长度为 4 的计算链：

```

public static Subquery chainsLength4(Object pairs) {
    return new Subquery("?a", "?b", "?c", "?d")
        .predicate(pairs, "?a", "?b")
        .predicate(pairs, "?b", "?c")
        .predicate(pairs, "?c", "?d");
}

```

为了使这个程序通用化，以找到任意长度的计算链，你需要一个生成正确数量的谓词和变量的子查询的函数。这可以通过编写一些很简单的 Java 代码来实现：

```

public static Subquery chainsLengthN(Object pairs, int n) {
    List<String> genVars = new ArrayList<String>();
    for(int i=0; i<n; i++) {
        genVars.add(Api.genNullableVar());
    }
    Subquery ret = new Subquery(genVars);
    for(int i=0; i<n-1; i++) {
        ret = ret.predicate(pairs, genVars.get(i), genVars.get(i+1));
    }
    return ret;
}

```

生成唯一的可为空值的输出变量。

循环来定义所需数量的连接。

该函数的一个有趣之处在于，转发微博数据不是特定的：可以将任意子查询或包含微博标识符对的数据源作为输入，并返回计算链的子查询。

再来看一个动态创建子查询的示例。假设要从未知大小的数据集中得到 N 个元素的随机样本。最简单的以分布式和可扩展的方式实现这个示例的策略是如下的算法：

- 1) 为每个元素生成一个随机数。
- 2) 找出随机数最小的 N 个元素。

JCascalog 有一个内置的名为“Limit”的聚合器，可以用来执行第二步。Limit 使用和并行聚合器类似的策略，它在每个 map 任务中找出最小的 N 个元素，然后合并所有 map 任务中的结果来得到整体最小的 N 个元素。下面的代码实现了这个能得到随机样本的策略：

```

public static Subquery fixedRandomSample(Object data, int n) {
    List<String> inputVars = new ArrayList<String>();
    List<String> outputVars = new ArrayList<String>();
    for(int i=0; i < Api.numOutFields(data); i++) {
        inputVars.add(Api.genNullableVar());
        outputVars.add(Api.genNullableVar());
    }

    String randVar = Api.genNullableVar();
    return new Subquery(outputVars)
        .predicate(data, inputVars)
        .predicate(new RandLong(), randVar)
        .predicate(Option.SORT, randVar)
        .predicate(new Limit(n), inputVars).out(outputVars);
}

```

通过自查输入数据集来确定输入和输出字段的正确数量。

为输入和输出变量生成独立的字段。

创建一个独立的字段来保存随机值。

使用 JCascalog-RandLong 函数将随机值追加到每个输入元组中。

对随机值执行二次排序。

使用 Limit 聚合器从数据集中找出 N 个随机元组。

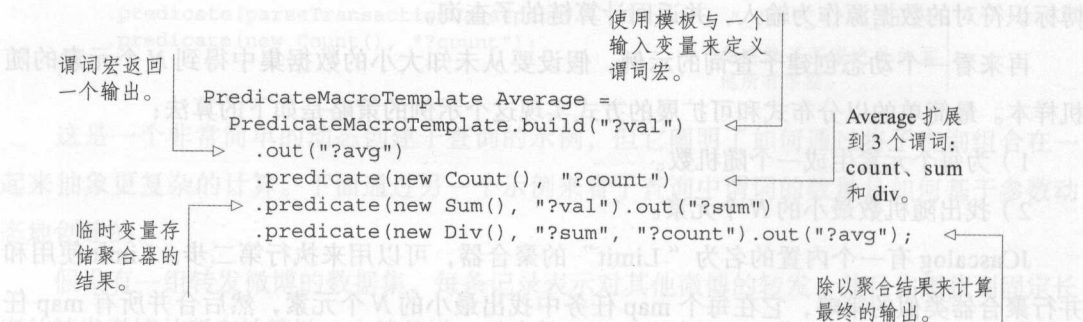
这种算法具有较高的可扩展性：它对固定的样本进行并行化计算，而不需要将所有记录集中在一个地方。

当编写 JCascalog 查询时，你将注意到，某些谓词的组合是经常一起使用的。在这种情况下，使用单个操作来表达集合体的功能将更加简单、高效。下面将深入研究 JCascalog 如何通过使用谓词宏来支持这种能力。

7.4.3 谓词宏

谓词宏是 JCascalog 扩展到谓词的不同集合的操作。因为 JCascalog 将所有操作表示为谓词，所以无论谓词是聚合器、过滤器还是函数，谓词宏都可以通过将这些谓词组合在一起创建强大的抽象。

你已经在本章的开始看到了定义 `Average` 的谓词宏的一个示例。该定义如下：



`Average` 包括3个组合在一起的谓词：计数聚合器、求和聚合器和一个除法函数。

图 7-17 展示了如何调用 `Average` 以及其产生的扩展。

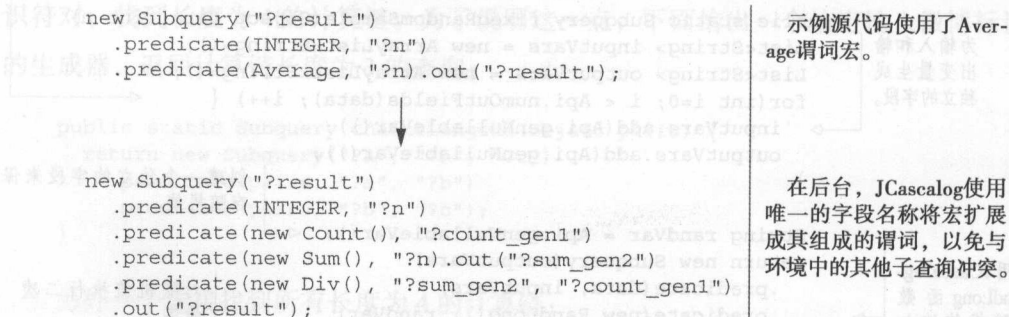
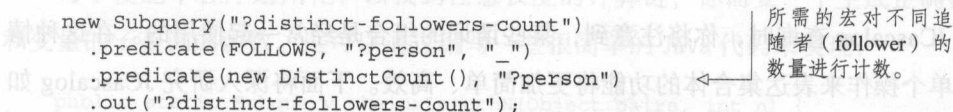


图 7-17 谓词宏为编写 JCascalog 自动扩展到组成谓词的简单查询提供了强大的抽象

`Average` 的定义是使用 JCascalog 模板来指定谓词宏应该扩展到的具体的谓词。但是，并非所有谓词都可以用模板指定。例如，假设你想创建一个谓词宏来计算给定的变量集中不同值的数量，代码如下：



该子查询确定了至少关注了另外一个人的不同用户的数量。与计算一个变量的平均值不同，你可以计算任意大小的变量集的不同计数。你不能使用模板，因为模板只支持固定的输入集和输出变量。下面来看能实现该功能的定义谓词宏的另一种方式。

首先，你需要定义一个执行实际计算的聚合器。即使一个组中元组的数量太大以至于不能包含在内存中，该聚合器也必须能够工作。为了解决这个问题，你可以利用二次排序特性在元组被聚合器处理之前对其进行排序。排序一旦完成，只有当前元组不同于其前一

个元组时，聚合器才增加不同的计数。

执行聚合的代码如下：

```
public static class DistinctCountAgg extends CascalogAggregator {
    static class State {
        int count = 0;
        Tuple last = null;
    }

    public void start(FlowProcess process, AggregatorCall call) {
        call.setContext(new State());
    }

    public void aggregate(FlowProcess process, AggregatorCall call) {
        State s = (State) call.getContext();
        Tuple t = call.getArguments().getTupleCopy();
        if(s.last==null || !s.last.equals(t)) {
            s.count++;
            s.last = t;
        }

        public void complete(FlowProcess process, AggregatorCall call) {
            State s = (State) call.getContext();
            call.getOutputCollector().add(new Tuple(s.count));
        }
    }
}
```

初始化每个组的追踪状态。

追踪当前计数和前一个出现的元组的内部状态。

当处理一个元组时，首先获取其当前状态。

只有当前元组不同于前一个元组时，才增加不同计数。

只更新状态中最近出现的元组。

若组中的所有元组被处理完，则生成不同计数。

`DistinctCountAgg` 包含对给定已排序的输入计算唯一计数的逻辑；不出所料，`JCascalog` 包含一个用于指定如何对每个组的元组进行排序的 `Option.SORT` 谓词。下面的代码展示了如何定义排序和手动计算不同的计数：

```
public static Subquery distinctCountManual() {
    return new Subquery("?distinct-followers-count")
        .predicate(FOLLOWS, "?person", "_")
        .predicate(Option.SORT, "?person")
        .predicate(new DistinctCountAgg(), "?person")
        .out("?distinct-followers-count");
}
```

根据“?person”字段对元组排序

当然，你会更喜欢这里的谓词宏，这样每次想做一个不同的计数时，就不必指定排序和聚合器了。谓词宏最普遍的形式是函数，它需要一个输入字段列表和输出字段列表，然后返回一组谓词。下面是将 `DistinctCount` 作为普通的 `PredicateMacro` 的定义：

```
public static class DistinctCount implements PredicateMacro {
    public List<Predicate> getPredicates(Fields inFields,
                                         Fields outFields) {
        List<Predicate> ret = new ArrayList<Predicate>();
    }
}
```

当在子查询中使用该宏时，输入和输出字段就确定了。

```

ret.add(new Predicate(Option.SORT, inFields));
ret.add(new Predicate(new DistinctCountAgg(),
                      inFields,
                      outFields));
return ret;
}

```

根据提供的输入字段对组排序。

对于该宏，不同的计数生成一个字段，但一般的宏形式支持多个输出。

7.4.4 动态创建谓词宏

你之前看到了普通 Java 函数可以动态地创建子查询，所以对于谓词宏也可以这么做应该没什么好惊讶的。这是一项非常强大的技术，它展示了让数据处理工具成为通用编程语言库的优点。

请思考下面的查询：

```

new Subquery("?x", "?y", "?z")
  .predicate(TRIPLETS, "?a", "?b", "?c")
  .predicate(new IncrementFunction(), "?a").out("?x")
  .predicate(new IncrementFunction(), "?b").out("?y")
  .predicate(new IncrementFunction(), "?c").out("?z");

```

读取包含三元组数值的数据集。

返回每个字段被增量的新的三元组。

虽然这是一个简单的查询，但有相当多的重复，因为它必须显式地给输入数据中的每个字段应用 IncrementFunction。最好能够消除这种重复，类似这样：

```

new Subquery("?x", "?y", "?z")
  .predicate(TRIPLETS, "?a", "?b", "?c")
  .predicate(new Each(new IncrementFunction()), "?a", "?b", "?c")
  .out("?x", "?y", "?z");

```

Each 谓词宏不是反复使用 IncrementFunction，而是将函数应用到指定的输入字段，并生成所需的输出。谓词宏的扩展与原始查询中 3 个独立的谓词相匹配。Each 谓词宏的代码如下：

```

public static class Each implements PredicateMacro {
    Object _op;

    public Each(Object op) {
        _op = op;
    }

    public List<Predicate> getPredicates(Fields inFields,
                                         Fields outFields) {
        List<Predicate> ret = new ArrayList<Predicate>();
        for(int i=0; i<inFields.size(); i++) {
            Object in = inFields.get(i);

```

Each 通过要使用的谓词操作被参数化。

```

Object out = outFields.get(i);
ret.add(new Predicate(_op,
    Arrays.asList(in),
    Arrays.asList(out)));
}
return ret;
}
}

```

谓词宏为每个给定的输入/输出字段对创建了一个谓词。

再来看另一个动态谓词宏的例子。我们之前将 `IncrementFunction` 定义为它自己的用来增量参数的函数，但实际上它只是将参数设置为 1 的 `Plus` 函数。拥有抽象谓词操作的部分应用程序的谓词宏是很有用的。然后你可以采用如下代码来定义 `Increment` 操作：

```
Object Increment = new Partial(new Plus(), 1);
```

正如你所看到的，`Partial` 是需要填写一些输入字段的谓词宏。它允许重写增量三元组的查询，代码如下：

```

new Subquery("?x", "?y", "?z")
    .predicate(TRIPLETS, "?a", "?b", "?c")
    .predicate(new Each(new Partial(new Plus(), 1)), "?a", "?b", "?c")
    .out("?x", "?y", "?z");

```

扩展完所有谓词宏之后，该查询转换为如下样式：

```

new Subquery("?x", "?y", "?z")
    .predicate(TRIPLETS, "?a", "?b", "?c")
    .predicate(new Plus(), 1, "?a").out("?x")
    .predicate(new Plus(), 1, "?b").out("?y")
    .predicate(new Plus(), 1, "?c").out("?z");

```

`Partial` 的定义很简单：

```

public static class Partial implements PredicateMacro {
    Object _op;
    List<Object> _args;

    public Partial(Object op, Object... args) {
        _op = op;
        _args = Arrays.asList(args);
    }

    public List<Predicate> getPredicates(Fields inFields,
                                         Fields outFields) {
        List<Predicate> ret = new ArrayList<Predicate>();
        List<Object> input = new ArrayList<Object>();
        input.addAll(_args);
        input.addAll(inFields);
        ret.add(new Predicate(_op, input, outFields));
        return ret;
    }
}

```


当子查询被创建时，谓词宏只是将任意提供的输入字段预先规划到指定的输入字段。正如你所看到的，动态的谓词宏对处理子查询的结构非常有效。

7.5 总结

如果想避免复杂性、防止错误和提高效率，那么表达计算的方式是至关重要的。对抗复杂性的主要技术是抽象和组合，并且数据处理工具要支持这些技术而不是让它们变得更困难，这是很重要的。

第8章和第9章将通过构建 SuperWebAnalytics.com 的批处理层，来强化批处理层的概念。SuperWebAnalytics.com 是一个更为复杂的现实示例，目的是在结构、算法和实现方面真正展示批处理计算的复杂性。

Each 谓词宏不是反复使用 `forEach` 来遍历输入数据，而是将输入数据放入数组，并生成所需的输出。谓词宏的扩展与原始查询中 3 个独立的谓词用新的谓词宏的代码如下：

```
public partial Object op, Object... args) {
    op = op;
    args = Arrays.asList(args);

    public List<Predicate> getPredicates(Predicate... preds) {
        List<Predicate> res = new ArrayList<Predicate>();
        List<Object> input = new ArrayList<Object>();
        input.addAll(args);
        input.addAll(preds);
        res.addAll(predicates);
        return res;
    }
}
```

批处理层示例：架构和算法

本章内容

- 从头至尾地创建一个批处理层
- 预先计算的应用示例
- 迭代图算法
- 用于高效集合基数操作的 HyperLogLog

你现在已经了解了批处理层的所有方面：为数据制定模式、存储主数据集以及以最小的复杂性运行大规模计算。本章将把这些方面拼凑起来，组成一个连贯的批处理层。本章没有介绍新的理论——旨在通过从头至尾地查看批处理层的设计，来强化前面章节提及的概念。这对于理解如何从理论映射到重要示例具有巨大的价值。

具体来说，你将学习如何创建批处理层来运行 SuperWebAnalytics.com。SuperWebAnalytics.com 足够复杂，它需要一个相当烦琐的批处理层，但不至于复杂到使你迷失在细节之中。

你会发现各种各样的批处理层抽象配合得很好，并且最终生成的 SuperWebAnalytics.com 批处理层是很优雅的。

在回顾 SuperWebAnalytics.com 的产品需求之后，我们将为“批处理层必须实现什么”“每个批处理视图应该预先计算什么”提供一个广泛的概述。你会在本章看到批处理层的体系结构和算法（使用管道图），会在第9章看到使用具体工具编写代码来实现它们的过程。

在本章中，请记住批处理层的灵活性。本章涉及 3 个示例批处理视图的工作流，但批处理层很容易扩展到计算新视图。这意味着“准备好了适应不断变化的客户需求和应用程序需求”是批处理层固有的特性。

8.1 SuperWebAnalytics.com 批处理层的设计

你将构建 SuperWebAnalytics.com 的批处理层，来支持 3 种类型查询的计算。回想一下，由于批处理层的目标是预先计算视图，因此指定的查询可以满足低延迟。介绍完 SuperWebAnalytics.com 将支持的查询之后，我们将讨论响应这些查询所需的批处理视图。

8.1.1 所支持的查询

SuperWebAnalytics.com 将支持以下 3 种不同类型的查询：

- 按照时间切片基于 URL 的页面浏览计数——“去年每一天的页面浏览量是多少？”和“过去 12 小时中有多少页面浏览量？”
- 按照时间切片基于 URL 的独立访客——“2010 年有多少独立访客经常访问这个域名？”和“过去三天内的每个小时有多少独立访客访问该域名？”
- 跳出率分析——“用户访问该站点的某个页面，而没有访问其他任何页面的百分比是多少？”

人们建模的方式使得第二类查询更具挑战性。回想一下，SuperWebAnalytics.com 模式用登录用户的用户 ID 或浏览器的 Cookie 标识符来表示一个人。因此一个人可以用不同的标识符访问同一网站——如果清除 Cookie，那么他们的 Cookie 可能改变，或者用户可以注册多个用户 ID。

该模式通过定义等效边来处理这种多样性。等效边表明两种不同的用户表现形式实际上代表的是同一个人。一个人的等效图可以是任意复杂的，如图 8-1 所示。准确地计算第二类查询要求你必须分析数据，以确定哪些页面浏览属于使用不同标识符的同一人。

8.1.2 批处理视图

接下来，我们将回顾满足每个查询所需要的批处理视图。每个批处理视图的关键在于预先计算的视图的大小和查询时所需的动态计算总量之间的平衡。

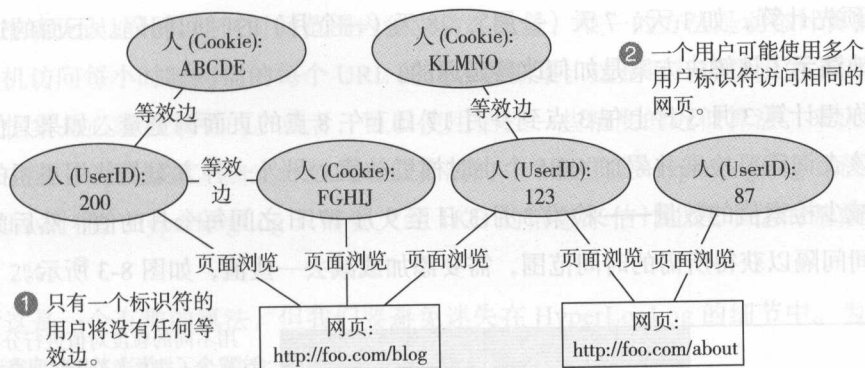


图 8-1 同一个人被捕获到使用不同标识符的不同页面浏览的示例

给定时间范围内的页面浏览

你希望能够检索细至小时粒度的任何时间范围内的一个 URL 的页面浏览量。就像第 4 章所提及的，预先计算每个可能的时间范围内的页面浏览量是不可行的，因为数据集每年涉及的每个 URL 需要预先计算难以管理的 3.8 亿个值。相反，你可以预先计算较少的数量，但在查询时需要更多的计算。

最简单的方法是为每个小时桶的每个 URL 预先计算页面浏览量。这将产生如图 8-2 所示的批处理视图。要解决一个查询，你需要检索其时间范围内每个小时桶的值，并把这些值相加。

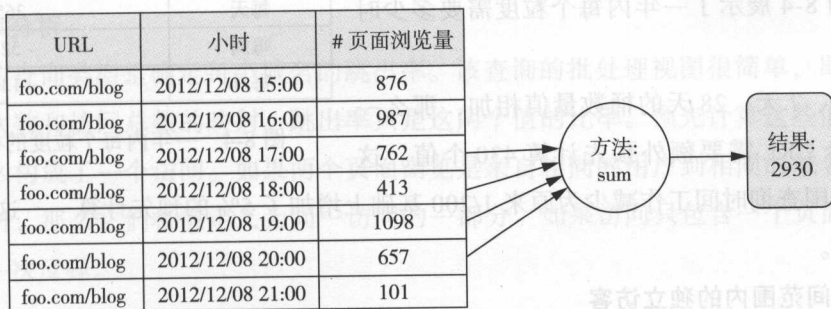


图 8-2 预先计算小时粒度的页面浏览量

但是该方法有一个问题——随着时间范围的大小增加，该查询会变得越来越慢。若要找出一年中的页面浏览量，则需要从批处理视图中检索大约 8760 个值，并加在一起。因为很多这些值将从磁盘被检索到，这可能会导致大范围查询的延迟，这实质上高于小范围查询的延迟。

幸运的是，解决办法很简单。不仅可以使用时粒度来预先计算值，也可以用更粗的

粒度进行预先计算，如 1 天、7 天（一周）、28 天（一个月）的时间间隔。下面的这个示例可以很好地演示了该解决方案是如何改善延迟的。

假设你想计算 3 月 3 日上午 3 点到 9 月 17 日上午 8 点的页面浏览量。如果只使用每小时的值，该查询需要检索并累加 4805 个小时桶里的值。另外一种方案是使用更粗的粒度可以显著地减少检索值的数量——检索 3 月 3 日至 9 月 17 日之间每个月的值，然后为了进一步精确时间间隔以获得所需的时间范围，需要添加或减去一些值，如图 8-3 所示。

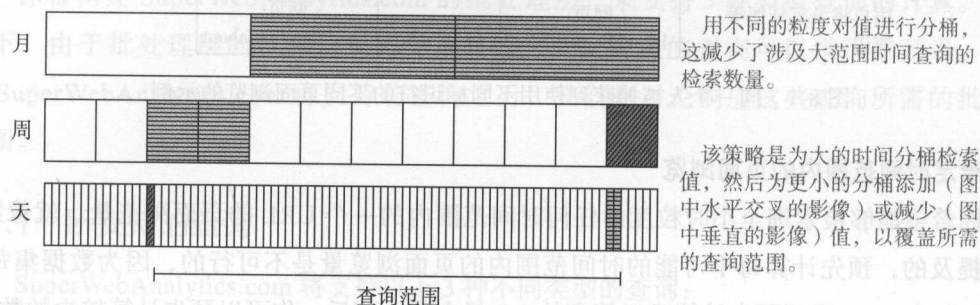


图 8-3 使用粗粒度来优化大范围查询的页面浏览量

对于该查询，只需要检索 26 个值——几乎减少为原来的 1/200！你可能想知道除了小时桶之外，预计算 1 天、7 天、28 天间隔的值有多昂贵。令人惊讶的是，这几乎没有任何额外的成本。图 8-4 展示了一年内每个粒度需要多少时间桶。

粒度	一年内桶的数量
每小时	8760
每天	~ 365
每周	~ 52
每月	~ 13

图 8-4 一年内每个粒度的桶数量

把 1 天、7 天、28 天的桶数量值相加，那么一年内的每个 URL 需要额外预先计算 430 个值。这

只是在大范围查询时间工作减少为原来 1/200 基础上增加了 5% 的预先计算——这是可接受的折衷方案。

给定时间范围内的独立访客

下一类查询旨在确定指定时间间隔内的独立访客数量。这看起来应该类似于给定时间范围内的页面浏览量，但是有一个关键的区别——独立计数不是加法。可以通过将单独每个小时的值进行累和来得到两个小时范围的页面浏览量，但不能对该查询类型做同样的操作。这是因为独立计数代表了元素集合的大小，而且每小时的集合之间可能有重叠。如果只是将两个小时的计数加在一起，那么会重复计算这两个时间间隔内都访问过该 URL 的人。

要想准确无误地计算任何时间范围内独立访客数量，唯一的方法是动态计算独立计数。这需要随机访问每小时时间桶的每个 URL 的访客数据集。这是可行但昂贵的，因为实质上整个主数据集是必须被索引的。或者，可以使用牺牲一些精度的近似算法，来大大减少批处理视图中索引的数据量。独立计数的近似算法的一个例子是 HyperLogLog 算法。对于每个 URL 和小时桶，HyperLogLog 只需要大约 1KB 的信息来估计上至十亿的基数集，最大错误率为 2%。^①

尽管这是一个有趣的算法，但我们要避免迷失在 HyperLogLog 的细节中。为避免这种情况，让我们把它当作一个黑盒子，关注它的接口即可：

```
interface HyperLogLog {
    long size();
    void add(Object o);
    HyperLogLog merge(HyperLogLog... otherSets);
}
```

每个 HyperLogLog 对象表示元素的一个集合，并支持添加新的元素到集合、合并其他 HyperLogLog 集以及检索集合的大小等操作。使用 HyperLogLog 使得给定时间范围内的独立访客查询与给定时间范围内的页面浏览量查询非常相似。关键的不同之处在于，要为每个 URL 和时间桶计算一个相对较大的值，并且 HyperLogLog 合并函数是用来合并时间桶而不是加和计数。与给定时间范围内的页面浏览量一样，需要通过创建 1 天、7 天、28 天粒度的 HyperLogLog 集合来减少查询时要做的工作量。

跳出率分析

最后的查询类型是确定每个域名的跳出率。该查询的批处理视图很简单，即从每个域名到反弹次数和访问总数的映射。跳出率只是这两个值的比率。预先计算这些值的关键是要明确什么构成了一个访问。如果两个页面浏览是来自相同的用户到相同的域名并且相距不到半小时，那么它们将被定义为同一访问的一部分。如果访问只包含一个页面，那么它被认为是一次反弹。

8.2 工作流概述

既然已经理解了批处理视图的具体要求，我们就可以在高层次上定义批处理层工作流。

① HyperLogLog 算法在标题为“HyperLogLog: the analysis of a nearoptimal cardinality estimation algorithm”的研究报告中有所描述，作者是 Flajolet, Éric Fusy, Olivier Gandouet 和 Frédéric Meunier，可以在 <http://algo.inria.fr/flajolet/Publications/FIFuGaMe07.pdf> 查找到。

工作流的基本原理如图 8-5 所示。

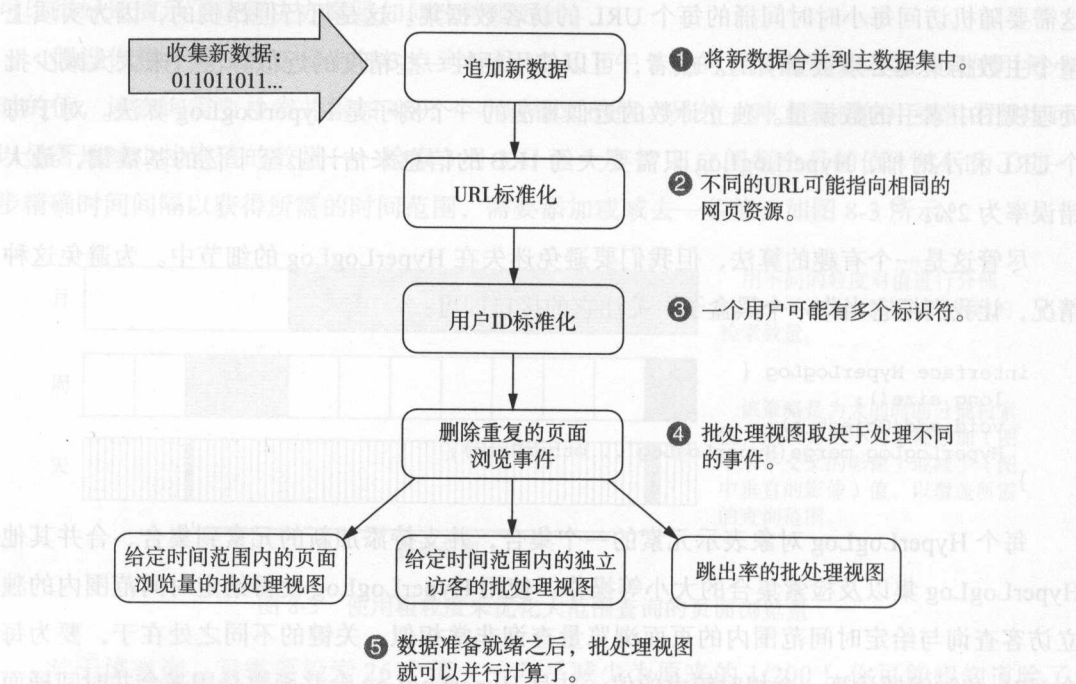


图 8-5 SuperWebAnalytics.com 的批处理工作流

批处理层工作流的起始端是分布式文件系统上包含主数据集的一个文件夹。第一步只是获取自从上次运行批处理层以来积累的所有新数据，并将它追加到主数据集中。

接下来的两个步骤是规范化数据，为计算批处理视图做准备。第一个规范化步骤使不同的 URL 可以指向相同的资源。例如，不同的 URL `www.mysite.com/blog/1?utm=1` 和 `http://mysite.com/blog/1` 指向相同的位置。第一个规范化步骤将所有 URL 转换成标准格式，使以后的计算可以正确地聚合数据。

第二个规范化步骤是必要的，因为同一个人的数据可能存在于不同的用户标识符下。为了支持关于访问和访客的查询，你必须为每个人选择单个标识符。后一种规范化步骤处理等效图以完成这项任务。因为批处理视图只利用页面浏览数据，所以只有页面浏览的边将被转换来使用这些被选择出的用户 ID。

下一个步骤是删除重复的页面浏览事件。回顾第 2 章中所提及的“通过让数据单元包含足够的信息使其具备唯一标识”的优势。在有问题的情况下（如网络分区），多次注册相同的页面浏览以确保事件被记录是常见的。删除重复的页面浏览对于计算批处理视图是必要的，因为它们依赖于数据集中不同的事件。

最后一步是使用规范化的数据计算在前一节中描述的批处理视图。**注意：**该工作流是纯粹的重新计算工作流——每次添加新数据，批处理视图都从头开始重新计算。在后面的章节中，你将知道在许多情况下可以增量化批处理层，以至于使用整个主数据集重新计算并不总是必需的。但定义纯粹的重新计算工作流绝对是必要的，因为当视图被损坏时，你需要从头开始重新计算。

现在让我们看看设计的每个详细步骤。我们将关注架构和算法，展示每个数据转换步骤的管道图，并在下一章讨论详细的代码细节。

8.3 获取新数据

一种你可能采取的，用来添加数据到主数据集的方法是——随着新数据的到来，将新文件插入主数据集文件夹中。但是该方法有一个问题，假设批处理工作流需要在主数据集上运行多个计算，如计算多个视图。这些计算可能在不同的时间开始，这意味着每个视图将代表不同时期的主数据集。然而这并不一定是致命的弱点，当你知道这些计算是基于完全相同的主数据集时，我们认为这对于你思考视图来说就简单多了。

处理该问题的一种简单方法是将新数据写入“new-data/”文件夹，然后批处理工作流的第一步是将任何在“new-data/”下的数据移动到主数据集文件夹（该过程可能会对数据垂直分区）。一旦数据被移动，就删除“new-data/”中对应的文件。这样做的结果是当数据被添加到主数据集时，批处理工作流有完全的控制权，并确保每个批处理视图是基于完全相同的主数据集。

8.4 URL 规范化

工作流的下一个步骤是规范化主数据集中的所有 URL。完成这项任务的查询需要一个实现了规范化逻辑的自定义函数。规范化包括剥离 URL 参数、在开始添加“http://”、删除末尾斜杠等。最重要的是，所有标准化逻辑都内置在一个函数中，并且可以独立操作每一个 URL。

该计算的管道图非常简单，如图 8-6 所示。正如你所看到的，所有它必须做的就是每个数据单元上运行该函数。

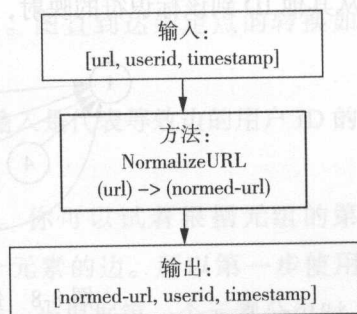


图 8-6 URL 规范化的管道图

从管道图的对象中提取字段

一般来说, 数据被封装到包含你所关心字段的对象中。例如, Pageview 对象包含 url、timestamp 和 userid 字段。当使用实际实现的管道图时, 计算开始于一个包含对象的字段, 并且通过在该对象上运行函数来提取所欲控制的字段 (用来连接、分组、用作函数的参数等)。为了保持简洁, 在这一章里, 我们通常会跳过提取步骤, 即管道图始于组成被控制对象的字段的输入。

8.5 用户标识符规范化

下一个步骤是为每个人选择一个用户标识符。这是工作流中最复杂的一部分, 因为它涉及一个完全分布式迭代图算法。尽管它很复杂, 但只需要几个很小的管道图就能解决这个问题。使用适当的工具, 你可以只用大约 100 行代码就能实现该步骤 (将在下一章进行展示)。

通过等效边标记的用户 ID 属于同一个人。如果从数据集中可视化这些边, 你会看到许多独立的子图, 如图 8-7 所示。

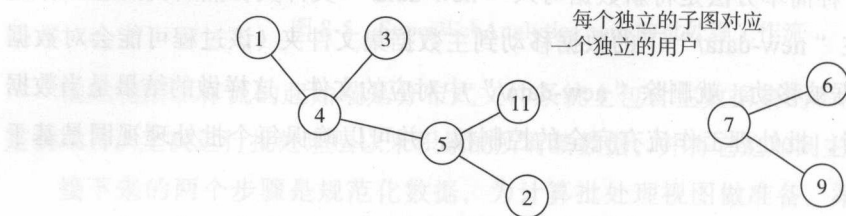


图 8-7 等效图示例

每个子图代表一个独立的用户。对于每一个人, 你需要选出一个标识符, 并创建一个从其他 ID 到该标识符的映射, 如图 8-8 所示。

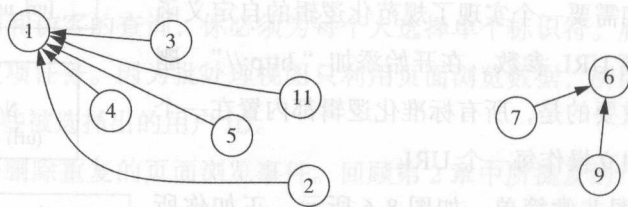


图 8-8 每个集合中从用户 ID 到单个标识符的映射

你可以通过将最初的等效图转换成如图 8-8 所示的形式来实现上述映射。对于该示例,

图 8-9 展示了它的转换，在该转换中，与某个人相关的每个用户 ID 都映射到为该人选出的一个唯一的 ID。该想法必须翻译成一个使用批处理计算可扩展运行的具体算法。之前所有批处理计算的示例均为一次执行单个管道图来生成所需的输出。然而对于该算法，要在一个步骤里得到所需的结果是不可能的。你必须采取迭代的方式，在每一步中修改图，以使图的状态接近所期望的结构（见图 8-9）。一旦定义完迭代步骤，它将被反复执行，直到没有进一步的进展。这就是所谓的“达到了定点”，此时生成的输出与输入相同，此时图已经达到了期望的状态。

用户ID	映射的用户ID
2	1
3	1
4	1
5	1
11	1
7	6
9	6

- ① 如果一个用户有多个ID，那么把这些ID映射到用户ID的最小值。
- ② 只需要存储映射之后的ID，所以省略了选择的ID(1,6)。

图 8-9 转换原始等效图，使集合中的所有节点都指向一个节点

在每次迭代中，该算法将检查图中每个节点的邻节点。它将确定连接的节点中最小的 ID，然后确保每条边都以最小值指向该节点。图 8-10 所示为修改单个节点周围的边的过程。

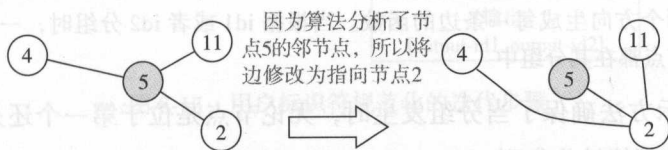


图 8-10 在一次迭代中修改单个节点周围的边

你可以在图 8-7 中看到等效图上的算法是如何工作的。图直达到定点的转换如图 8-11 所示。

让我们制定一个实现了上述迭代算法的管道图。算法的输入是代表等效边的用户 ID 的二元组集合。

迭代算法的首要需求是确定每个节点的直接邻节点。你可以试着根据元组的第一个元素进行分组，但这将排除给定节点存储在最后一个元素的边。所以第一步使用 BidirectionalEdges 函数，来生成两个方向的每一条边。这样，当根据第一个元素分组时，你就可以得到存在节点的每一条边。BidirectionalEdges 函数的示意图如图 8-12 所示。

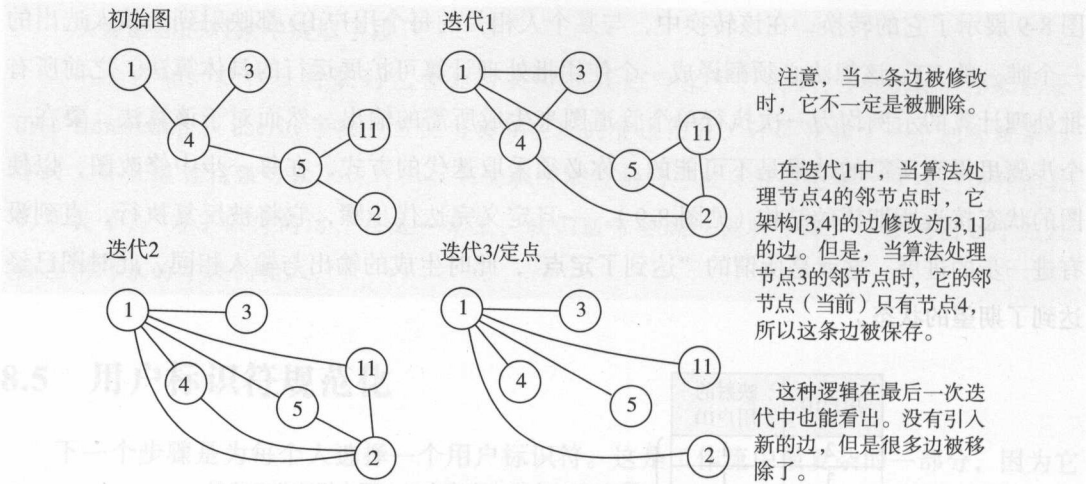


图 8-11 迭代该算法直到达到定点

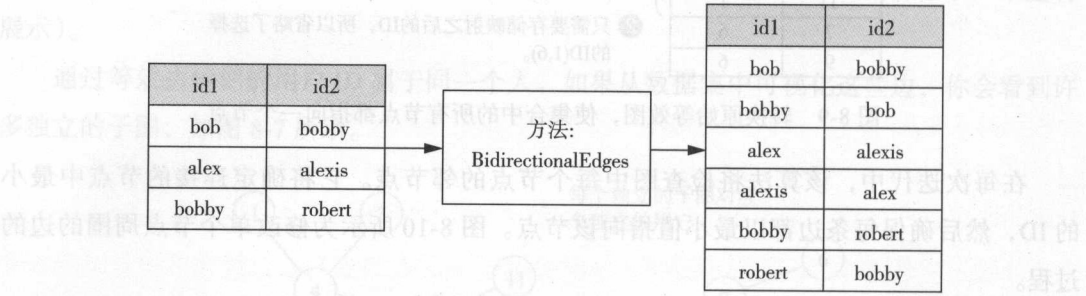


图 8-12 在两个方向生成每一条边的函数。当根据 id1 或者 id2 分组时，一个节点的所有邻节点都在其分组中

这种双重表示方法确保了当分组发生时，无论节点是位于第一个还是最后一个位置，包含节点的所有边都能被收集到。

现在对于每个边的分组，应该生成边的新集合，该集合中每个节点都指向组中最小的节点。这可以用一个简单的聚合器来完成，伪代码如下：

```
function userid-step-aggregator(grouped-node, edges) {
  nodes = new SortedSet()
  for(e in edges) {
    nodes.add(e.first)
    nodes.add(e.second)
  }
  target = nodes.smallest()
  isNewEdges = grouped != target && nodes.size() > 2
}
```

所有边的所有节点被收集到一个集合中。

选出最小的节点。

如果条件满足，至少生成一条新的边。该信息将用于确定何时停止运行迭代步骤。

```

for(n in nodes) {
  if(n != target) {
    emit(n, target, isNewEdges)
  }
}

```

除所选出节点的每个节点均生成一条到选出节点的新的边。

如果分组节点在其邻节点中最小，那么已经生成的边不变；否则，如果节点有多个邻节点，那么一些边会被修改为指向最小的标识符。若使用 BidirectionalEdges 函数和聚合器完成，则迭代步骤的管道图会很简单，如图 8-13 所示。

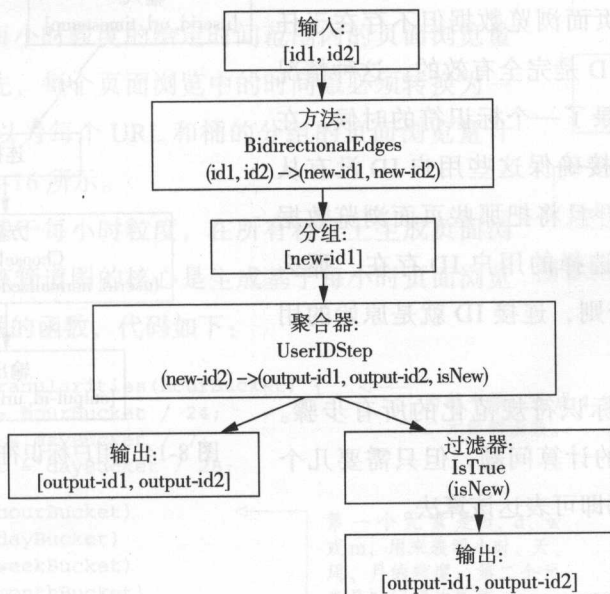


图 8-13 用户标识符规范化的迭代步骤

迭代步骤需要反复运行，直到没有边发生变化。迭代步骤的一个输出是边的新集合，另一个输出是不存在于步骤输入中的边的集合。一旦没有新的边生成，那么算法达到了一个定点，并且节点的连接集合中的每个节点都指向该集合中最小的节点。

为了完成这个算法，迭代步骤必须封装在一个循环中，直至达到定点，如下面的伪代码所示：

```

function userid-normalization(startingEdges) {
  isNewEdges = true
  while(isNewEdges) {
    [nextEdges, newEdges] = runNormalizationStep(startingEdges)
    isNewEdges = !newEdges.isEmpty()
  }
}

```

运行，直到没有新的边生成。

startingEdges 代表存储在计算集群中的边的集合。它没有在控制器程序中显式表示。

runNormalizationStep 覆盖之前的管道图，并返回边的新集合和全新的边。

实际上，这样的代码将运行在一台单独的机器上，而 `runNormalizationStep` 将调用一个作业在分布式计算集群上并行执行。当使用分布式文件系统存储输入和输出时，对于协调文件路径，多一些的代码是必要的，这些会在第 9 章中介绍。但这段代码涵盖了该算法的要点。

要完成该工作流，最后需要做的是，使用所选择的用户标识符来改变页面浏览数据中的用户 ID。这种转变可以通过执行页面浏览数据与等效图的最终迭代的连接来实现，如图 8-14 所示。

注意：存在于页面浏览数据但不存在于任何等效边中的用户 ID 是完全有效的。这种情况发生在用户只被记录了一个标识符的时候。在这些情况下，外连接确保这些用户 ID 没有从结果中被过滤掉，并且将把那些页面浏览数据连接到空值。如果选择的用户 ID 存在，那么它就是连接 ID；否则，连接 ID 就是原始的用户 ID。

以上就是用户标识符规范化的所有步骤。虽然这是一个更难的计算问题，但只需要几个管道图和一些伪代码即可表达该算法。

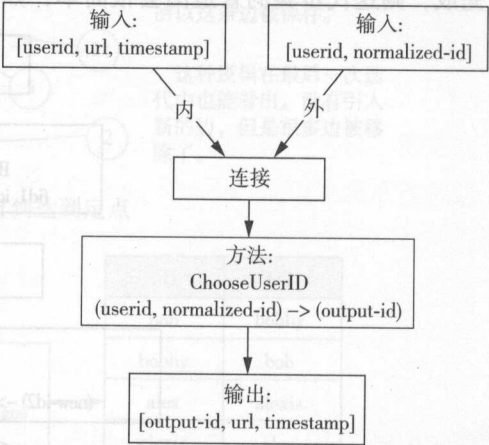


图 8-14 用户标识符规范化的最后步骤

8.6 页面浏览去重

计算批处理视图之前的最后准备步骤是删除重复的页面浏览事件。该管道图非常简单，如图 8-15 所示。

8.7 计算批处理视图

现在数据已经准备好计算批处理视图，就像本章开始设计的那样。该计算步骤将产生没有索引的记录；在后面的章节，你将学习如何索引批处理视图，这样它们就可以按随机访问的方式被查询到。

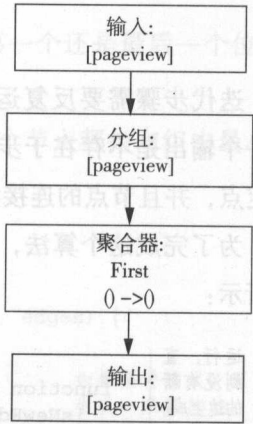


图 8-15 删除重复数据的管道图

8.7.1 给定时间范围内的页面浏览量

正如前面介绍的，给定时间范围内的页面浏览量的批处理视图，应该聚合每小时、每天、7天、28天粒度的每个URL的页面浏览量。你将采用的方法是：首先，聚合每小时粒度的页面浏览量（这将减少许多数量级的数据大小）；其次，你将加和每小时的值来得到更大的桶。由于输入的规模更小，因此后面的操作将快得多。

下面先从计算每小时粒度的给定时间范围内的页面浏览量的管道图开始。首先，每个页面浏览中的时间戳必须转换为一个小时桶，然后可以为每个URL和桶的分组的页面浏览量计数。该管道图如图8-16所示。

接下来看一下基于每小时粒度，在所有粒度上生成页面浏览计数的管道图。该管道图的核心是生成基于每小时页面浏览计数的每个粒度的桶的函数，代码如下：

```
function emitGranularities(hourBucket) {
  dayBucket = hourBucket / 24;
  weekBucket = dayBucket / 7;
  monthBucket = dayBucket / 28;

  emit("h", hourBucket)
  emit("d", dayBucket)
  emit("w", weekBucket)
  emit("m", monthBucket)
}
```

为每个输入生成4个二元组的函数。

第一个元素是h、d、w或m，用来表明小时、天、周、月的粒度；第二个元素是时间桶的数值。

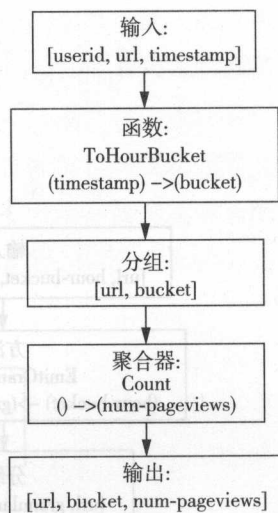


图 8-16 计算每小时粒度的给定时间范围内的页面浏览量

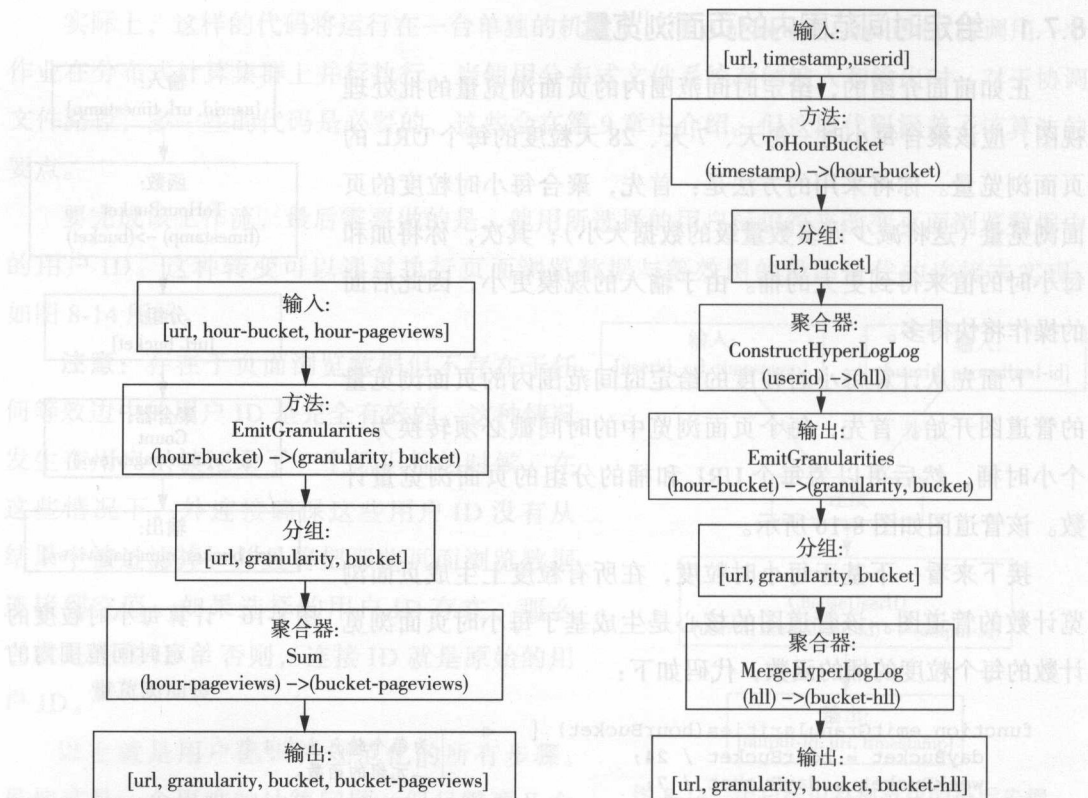
最后，管道图仅需相加每个URL/粒度/桶的页面浏览计数，如图8-17所示。

8.7.2 给定时间范围内的独立访客

给定时间范围内的独立访客的批处理视图包含追踪每个URL的每个时间粒度的HyperLogLog集。除了不是聚合计数而是聚合HyperLogLog集之外，它本质上与给定时间范围内的页面浏览量的计算是相同的。

计算小时HyperLogLog集以及更高粒度的HyperLogLog集的合并后的管道图如图8-18所示。

如图8-18所示，它只需要ConstructHyperLogLog和MergeHyperLogLog，这两个聚合器很容易编写。

图 8-17 所有粒度的给定时间范围内的
页面浏览量图 8-18 给定时间范围内的独立
访客的管道图

8.7.3 跳出率分析

最后的批处理视图计算每个 URL 的跳出率。就像本章开始概述的那样，你将每个域名计算两个值：访问总次数和反弹访问次数。

该查询的关键部分是追踪一个人浏览互联网所形成的每次访问。完成这项任务的一种简单的方法是，检查一个人对于一个特定域名的所有页面浏览，按时间顺序排序。然后使用连续页面浏览之间的时间差来确定它们是否属于相同的访问。如果一次访问只包含一个页面浏览，那么它视作一次反弹访问。

下面调用一个聚合器来完成 AnalyzeVisits。查看在一个域名上一个用户的所有页面浏览之后，AnalyzeVisits 生成两个字段：该域名上该用户的访问总数和反弹访问的次数。

跳出率分析的管道图如图 8-19 所示。这是一个更为复杂的计算，因为它需要多个聚合——但通过管道图来表示，这仍然是很容易的。

至此，SuperWebAnalytics.com 批处理层的工作流和算法就完成了。

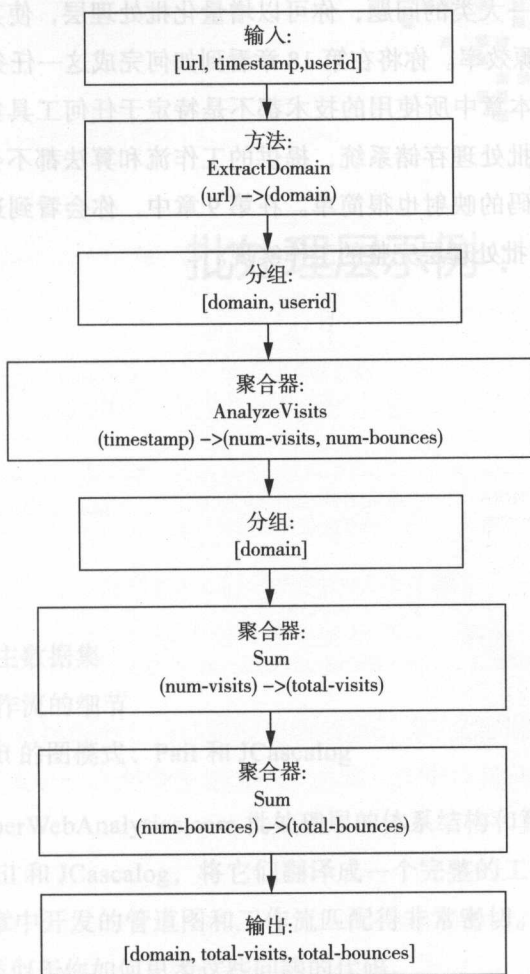


图 8-19 跳出率分析的管道图

8.8 总结

SuperWebAnalytics.com 的批处理层包含复杂的逻辑，但它很容易实现。这完全归根于批处理层在所有数据上计算函数的性质。当可以立即查看所有数据时，你没有受到增量算法限制的约束——构建系统既简单又容易。批处理计算也提供了极大的灵活性。通过扩展批处理层来计算新的视图是很容易的——工作流的每个阶段在所有数据上运行任意函数都是不受约束的。

实现工作流之前，让我们快速回顾到目前为止已经为 SuperWebAnalytics.com 开发了什

我们已经多次重申，本章开发的是基于重新计算的工作流，批处理视图总是从头开始重新计算。对该算法有一大类的问题，你可以增量化批处理层，使其不需要增加太多的复杂性就能获得更多的资源效率。你将在第 18 章看到如何完成这一任务。

我们想强调的是，本章中所使用的技术都不是特定于任何工具集的，所以无论你使用什么样的批处理计算和批处理存储系统，提供的工作流和算法都不会改变。在实践中，从工作流和算法到实际代码的映射也很简单。在第9章中，你会看到这些实际代码，并生成SuperWebAnalytics.com批处理层完整的工作实现。



批处理层示例：实现

本章内容

- 追加新数据到主数据集
- 管理批处理工作流的细节
- 整合基于 Thrift 的图模式、Pail 和 JCascalog

第8章介绍了 SuperWebAnalytics.com 批处理层的体系结构和算法。本章将之前介绍过的工具，如 Thrift、Pail 和 JCascalog，将它们翻译成完整的工作实现。在这个过程中，你会发现代码与第8章中开发的管道图和工作流匹配得非常密切。这表明抽象是合理使用的，因为你可以编写类似于你如何思考这些问题的代码。

就像现实生活中的工具都会发生的那样，你会遇到来自工具的人工复杂性的冲突。在这种情况下，你会看到源于 Hadoop 对小文件的限制的必然复杂性，而那些复杂性必须被解决。除了理解理想的工作流和算法，理解在实践中实现它们的具体细节也有巨大的价值。

第8章中开发的工作流如图9-1所示。不妨回顾一下，想想工作流的每个步骤都做了什么。

9.1 出发点

实现工作流之前，让我们快速回顾到目前为止已经为 SuperWebAnalytics.com 开发了什

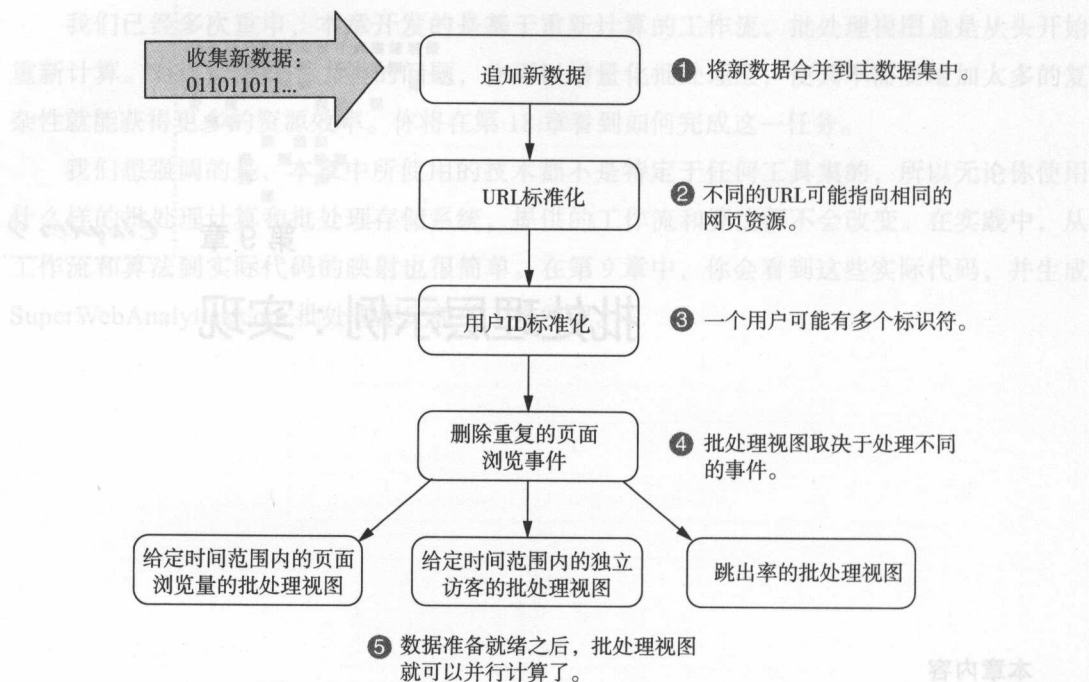


图 9-1 SuperWebAnalytics.com 的批处理工作流

么内容。到目前为止，你已经实现了批处理层的核心：一种表示数据模型的强大模式和将数据存储分布在分布式文件系统的能力。你用 Thrift 创建图模式——人和页面被表示为节点，页面浏览是人与页面之间的边，其他信息存储为节点的属性。

你使用 Pail 库与 HDFS 交互——Pail 为你提供了垂直分区数据、选择文件格式、管理诸如数据集之间的追加的基本操作的简单接口。创建 PailStructure 使得 SuperWebAnalytics.com 的 Data 对象被存储在 pail 内，根据边或属性的类型可选择地进行分区。

9.2 准备工作流

在开始实现工作流本身之前，你需要一个快速准备步骤。工作流的很多部分都要操作定义在 Thrift 模式中的对象，如 Data、PageViewEdge、PageID 和 PersonID 对象。Hadoop 需要知道如何对这些对象进行序列化和反序列化，以便在 MapReduce 作业中，这些对象可以在机器之间传输。

为了实现这一点，你必须为对象注册一个序列化器——cascading-thrift 项目 (<https://github.com/cascading/cascading-thrift>) 提供了一个适合该目标的实现。下面的代码片段演示

了如何为批处理 workflow 注册序列化器：

```
public static void setApplicationConf() {
    Map conf = new HashMap();
    String sers = "backtype.hadoop.thrift.serialization." +
        "org.apache.hadoop.io.serializer.WritableSerialization";
    conf.put("io.serializations", sers);
    Api.setApplicationConf(conf);
}
```

SuperWebAnaly-tics.com 对象的 Thrift 序列化器

Hadoop Writable 对象默认的序列化器

该代码指示 Hadoop 使用 Thrift 序列化器以及默认的 Hadoop 序列化器。当注册多个序列化器时，若 Hadoop 需要序列化一个对象，它将自动确定合适的序列化器。这段代码设置了全局配置，故批处理 workflow 中的每项作业都能使用。

有了这些方法，下面从获取新的数据开始实现 workflow。

9.3 获取新数据

在工作流的设计中，你看到了把传入的新数据和主数据集分隔开是何等重要。这可以防止当批处理 workflow 正在运行时新数据被插入主数据集中，避免独立的视图是略有不同的主数据集的表示的可能性。

所以工作流的第一步是获取 new-data/ 中的数据，将其添加到主数据集 pail，并从“new-data/”中删除该数据。虽然这在概念上很简单，但可能存在同步问题。暂时忽略实际追加的细节，假设你尝试以下的操作：

```
// do not use!
public static void badNewDataAppend(Pail masterPail, Pail newDataPail)
    throws IOException {
    appendNewDataToMasterDataPail(masterPail, newDataPail);
    newDataPail.clear();
}
```

这似乎很简单，但在这段代码中有一个隐含的竞争条件。在运行追加操作时，更多的数据可能被写入新数据 Pail 中。如果你在追加操作之后清除新数据 Pail，那么也将删除追加作业运行时写入的任何新数据。

幸运的是有一个简单的解决方案。Pail 提供了 snapshot 和 deleteSnapshot 方法来解决这个问题。snapshot 方法将 Pail 的快照存储在一个新的位置，而 deleteSnapshot 方法只从原始 pail 中删除存在于快照中的数据。使用这些方法，可确保只移除已经成功追加到主数据集 Pail 中的数据，代码如下：

在整个批处理工作流程中，/
tmp/swa被用作临时工作空间。

```
public static void ingest(Pail masterPail, Pail newDataPail)
    throws IOException {
    FileSystem fs = FileSystem.get(new Configuration());
    fs.delete(new Path("/tmp/swa"), true);
    fs.mkdirs(new Path("/tmp/swa"));

    Pail snapshotPail = newDataPail.snapshot("/tmp/swa/newDataSnapshot");
    appendNewDataToMasterDataPail(masterPail, snapshotPail);
    newDataPail.deleteSnapshot(snapshotPail);
}
```

获取新数据 Pail 的快照。

追加之后，只删除存在于快照中的数据。

从快照中追加数据到主数据集。

注意：这段代码还在 /tmp/swa 创建了临时工作空间。工作流的许多阶段都需要中间数据的空间，且在第一步执行之前初始化该暂存区域是非常合适的。

至此，本任务还未完成，因为我们必须看看实现 `appendNewDataToMasterDataPail` 函数的细节。`new-data/pail` 和主数据集 `pail` 之间的一个差异是，主数据集根据属性或边的类型进行垂直分区。`new-data/ pail` 只是新数据的一个堆放地，所以其中的每个文件中可能包含所有属性类型和边的数据单元。在数据可以被追加到主数据集之前，它必须先被重组，以符合主数据集 `Pail` 使用的结构。重组 `Pail` 以使其拥有一个新结构的过程被称为分解 (Shredding)。

为了分解一个 `Pail`，你必须能够通过 `JCasalog` 查询来写入和读取 `Pail`。在实现分解之前，让我们看看如何整合 `JCasalog` 和 `pail`。在 `JCasalog` 中，读写数据的抽象被称为 `tap`。`dfs-datastores` 项目 (<https://github.com/nathanmarz/dfs-datastores>) 提供了一个 `PailTap` 实现，因此 `Pail` 可以用作 `JCasalog` 查询的输入和输出。当用作源时，`PailTap` 检查 `Pail` 并自动反序列化它所包含的记录。

下面的代码创建了一个从 `Pail` 中读取所有数据作为查询来源的 `tap`：

```
public static void pailTapUsage() {
    Tap source = new PailTap("/tmp/swa/snapshot");
    new Subquery("?data").predicate(source, "_", "?data");
}
```

快照是一个 `SuperWebAnalytics.com Pail`，所以 `tap` 将生成 `Thrift Data` 对象。

`tap` 生成包含记录的文件和记录本身；工作流程中不需要文件名，所以文件名可以被忽略。

`PailTap` 还支持读取 `Pail` 内数据的一个子集。为了使用第 3 章中 `SplitDataPailStructure` 的 `Pail`，你可以构建一个读取只包含在 `Pail` 中的等效边的 `PailTap`：

```
PailTapOptions opts = new PailTapOptions();
opts.attrs = new List[] {
```

属性是列表的一个数组；每个列表包含作为输入的子文件夹的目录路径。

传递自定义配置到 `PailTap`。

```

        new ArrayList<String>() {{
            add("" + DataUnit._Fields.EQUIV.getThriftFieldId());
        }}
    };
    Tap equivs = new PailTap("/tmp/swa/snapshot", opts);

```

创建包含等
效边的相对
路径的列表。

创建带有指
定选项的
tap。

由此该功能经常会被用到，因此它应该被封装成一个函数以便后续使用：

```

public static PailTap attributeTap(String path,
                                   final DataUnit._Fields... fields) {
    PailTapOptions opts = new PailTapOptions();
    opts.attrs = new List[] {
        new ArrayList<String>() {{
            for(DataUnit._Fields field: fields) {
                add("" + field.getThriftFieldId());
            }
        }}
    };
    return new PailTap(path, opts);
}

```

可以指定多个子
文件夹作为 tap 的
输入。

当终端数据从查询到全新的 Pail 中时，你必须声明即将写入 PailTap 中的记录类型。你可以通过设置 spec 选项来包含合适的 PailStructure 完成这一任务。为了创建一个根据属性分解数据单元的 Pail，你可以使用第 5 章中的 SplitDataPailStructure：

```

public static PailTap splitDataTap(String path) {
    PailTapOptions opts = new PailTapOptions();
    opts.spec = new PailSpec((PailStructure) new SplitDataPailStructure());
    return new PailTap(path, opts);
}

```

现在你可以使用 PailTap 和 JCasalog 来实现工作流的分解部分。第一次尝试分解看起来是这样的：

```

// do not use!
public static void badShred() {
    PailTap source = new PailTap("/tmp/swa/snapshot");
    PailTap sink = splitDataTap("/tmp/swa/shredded");

    Api.execute(sink, new Subquery("?data").predicate(source, "_", "?data"));
}

```

该查询在逻辑上是正确的。但是尝试在 HDFS 海量的输入数据集上运行该查询时，你会遇到诸如 NameNode 错误、文件句柄限制等奇怪的问题。这些都是 Hadoop 本身的局限性。该查询的问题是，它创建了无数的小文件——就像第 7 章中讨论的那样，Hadoop 不能很好地处理大量的小文件。

要理解为什么会发生这种情况，你必须理解查询是如何执行的。因为该查询没有涉及聚合或连接，所以它只执行 map 作业，跳过了 reduce 阶段。这通常是非常可取的，因为

reduce 阶段是更昂贵的阶段。但是假设你的模式有 100 个不同的边和属性类型，那么一个 map 任务要创建 100 个独立的输出文件——一个文件存储一个记录类型。如果处理你的输入数据需要 10 000 个 mapper (大约 1.5 TB 的数据存储在 128-MB 的块中)，那么输出将包括大约一百万个文件，对于 Hadoop 来说，文件太多很难处理。

你可以通过将 reduce 阶段人为地引入计算中来解决这个问题。与 mapper 不同，你可以通过作业配置来显式地控制 reducer 的数量。如果你用 100 个 reducer 在 1.5 TB 的数据上运行这个假设的作业，会生成更易于管理的 10 000 个文件。下面的代码包含一个“个性聚合器”来强制查询执行 reduce 阶段：

```

注意：这
public static Pail shred() throws IOException {
    PailTap source = new PailTap("/tmp/swa/snapshot");
    PailTap sink = splitDataTap("/tmp/swa/shredded");

    Subquery reduced = new Subquery("?rand", "?data")
        .predicate(source, "_", "?data-in")
        .predicate(new RandLong())
        .out("?rand")
        .predicate(new IdentityBuffer(), "?data-in")
        .out("?data");

    Api.execute(sink,
        new Subquery("?data").predicate(reduced, "_", "?data"));

    Pail shreddedPail = new Pail("/tmp/swa/shredded");
    shreddedPail.consolidate();
    return shreddedPail;
}

```

为每条记录分配一个随机数。

在 reduce 阶段之后，项目输出随机数。

使用身份聚合器让每条数据记录进入 reducer。

合并分解的 Pail 来进一步减少文件的数量。

现在，数据已被分解，文件的数量也已经最小化，新数据终于可以被追加到主数据集 Pail 中了，代码如下：

```

public static void appendNewData(Pail masterPail,
    Pail snapshotPail) throws IOException {
    Pail shreddedPail = shred();
    masterPail.absorb(shreddedPail);
}

```

一旦新数据被追加到主数据集，你就可以着手规范化数据了。

9.4 URL 规范化

下一步是将主数据集中的所有 URL 规范化到它们的规范形式。虽然规范化涉及很多东西，如剥离 URL 参数、将 http:// 添加到开头以及删除末尾斜杠，但出于演示的目的，这里

仅提供基本的实现：

由于输入对象已被备份，所以它可以被安全地修改。

该函数获取 Data 对象并生成规范的 Data 对象。

```
public static class NormalizeURL extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        Data data = ((Data) call.getArguments().getObject(0)).deepCopy();
        DataUnit du = data.get_dataunit();

        if(du.getSetField() == DataUnit._Fields.PAGE_VIEW) {
            normalize(du.get_page_view().get_page());
        }
        call.getOutputCollector().add(new Tuple(data));
    }

    private void normalize(PageID page) {
        if(page.getSetField() == PageID._Fields.URL) {
            String urlStr = page.get_url();
            try {
                URL url = new URL(urlStr);
                page.set_url(url.getProtocol() + "://" + url.getHost()
                    + url.getPath());
            } catch (MalformedURLException e) {}
        }
    }
}
```

对于支持的批处理视图，只有页面浏览的边需要被规范化。

通过从 URL 中提取标准组件来规范化页面浏览数据。

你可以使用 NormalizeURL 函数创建一个规范化版本的主数据集。URL 规范化的管道图，如图 9-2 所示。

通过下面的代码将管道图翻译成 JCascalog：

```
public static void normalizeURLs() {
    Tap masterDataset = new PailTap("/data/master");
    Tap outTap = splitDataTap("/tmp/swa/normalized_urls");
    Api.execute(outTap,
        new Subquery("?normalized")
            .predicate(masterDataset, "_", "?raw")
            .predicate(new NormalizeURL(), "?raw")
            .out("?normalized"));
}
```

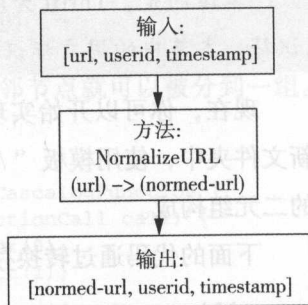


图 9-2 URL 规范化的管道图

9.5 用户标识符规范化

下面来实现工作流中最复杂的部分——用户标识符规范化。先来回顾一下，这是一个迭代图算法，操作如图 9-3 所示。

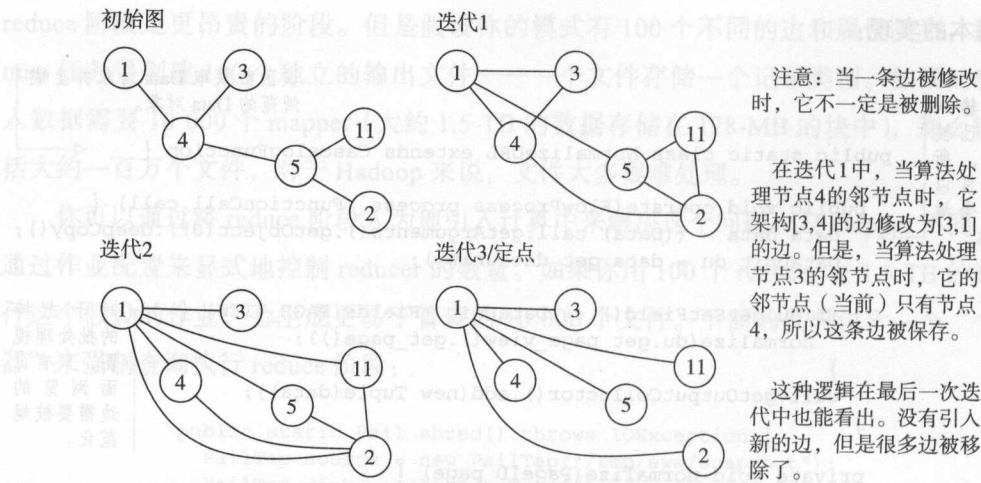


图 9-3 迭代该算法直至达到定点

对 Thrift 数据类型排序

你可能记得，PersonID 不是整数——实际上被建模为 Thrift 联合体：

```
union PersonID {
  1: string cookie;
  2: i64 user_id;
}
```

幸运的是，Thrift 为所有 Thrift 结构提供了固有的排序，可以用来确定“最小”的标识符。本节中的用户标识符规范化算法正是利用了 Thrift 的这个特性。

现在，你可以开始实现迭代算法。每个迭代的输出将被存储在分布式文件系统的一个新文件夹中，使用模板“/tmp/swa/equivs{ 迭代次数}”作为路径。这些输出将由 PersonID 的二元组构成。

下面的代码通过转换存储在主数据集中的等效边对象，来创建初始数据集：

```
public static class EdgifyEquiv extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    Data data = (Data) call.getArguments().getObject(0);
    EquivEdge equiv = data.get_dataunit().get_equiv();
    call.getOutputCollector()
      .add(new Tuple(equiv.get_id1(), equiv.get_id2()));
  }
}

public static void initializeUserIdNormalization() throws IOException {
  Tap equivs = attributeTap("/tmp/swa/normalized_urls",
    DataUnit._Fields.EQUIV);
  Api.execute(Api.hfsSeqfile("/tmp/swa/equivs0"),
    new Subquery("?node1", "?node2"))
```

一个用于从等效边中提取标识符的自定义函数。

初始化数据被存储为迭代 0 次。

```

        .predicate(equivs, "_", "?data")
        .predicate(new EdgifyEquiv(), "?node1", "?node2"));
    }

```

迭代步骤的管道图如图 9-4 所示。

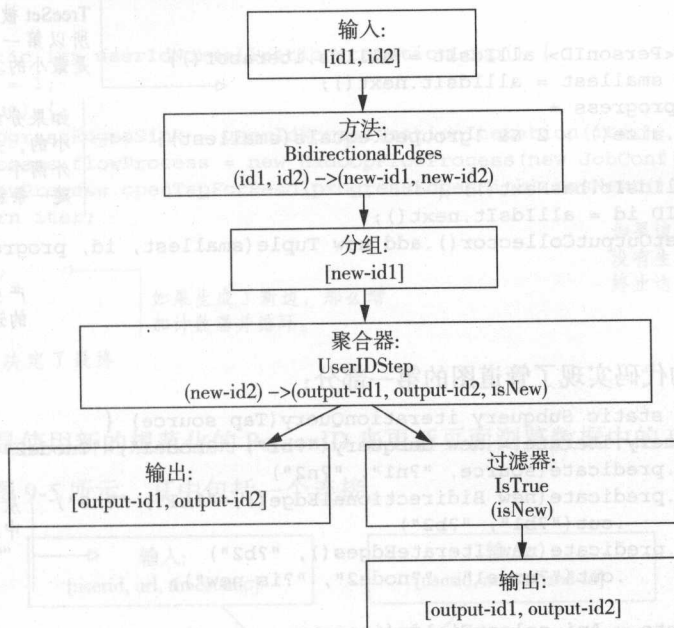


图 9-4 用户标识符规范化的迭代步骤的管道图

前面提到，边必须在两个方向上生成，这样一个节点的所有邻节点就可以被分到一组。

下面的自定义函数生成两个方向的边：

```

public static class BidirectionalEdge extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        Object node1 = call.getArguments().getObject(0);
        Object node2 = call.getArguments().getObject(1);
        if (!node1.equals(node2)) {
            call.getOutputCollector().add(new Tuple(node1, node2));
            call.getOutputCollector().add(new Tuple(node2, node1));
        }
    }
}

```

过滤掉任何连接到自身的边。

使用 [a, b] 和 [b, a] 的顺序生成边。

它们一旦被分组，你就需要一个自定义聚合器来实现算法逻辑，并表示出哪些边是新的：

```

public static class IterateEdges extends CascalogBuffer {
    public void operate(FlowProcess process, BufferCall call) {
        PersonID grouped = (PersonID) call.getGroup().getObject(0);
    }
}

```

使用节点
对元组进
行分组。

```

TreeSet<PersonID> allIds = new TreeSet<PersonID>();
allIds.add(grouped);

Iterator<TupleEntry> it = call.getArgumentsIterator();
while(it.hasNext()) {
    allIds.add((PersonID) it.next().getObject(0));
}

Iterator<PersonID> allIdsIt = allIds.iterator();
PersonID smallest = allIdsIt.next();
boolean progress =
    allIds.size() > 2 && !grouped.equals(smallest);

while(allIdsIt.hasNext()) {
    PersonID id = allIdsIt.next();
    call.getOutputCollector().add(new Tuple(smallest, id, progress));
}
}

```

TreeSet 包含节点及其所有邻节点。

TreeSet 被存储，所以第一个元素是最小的。

如果分组的节点不是最小的，并连接了至少另外两个节点，那么将创建一条新的边。

产生迭代期间生成的边。

最后，下面的代码实现了管道图的第一部分：

```

public static Subquery iterationQuery(Tap source) {
    Subquery iterate = new Subquery("?b1", "?node1", "?node2", "?is-new")
        .predicate(source, "?n1", "?n2")
        .predicate(new BidirectionalEdge(), "?n1", "?n2")
        .out("?b1", "?b2")
        .predicate(new IterateEdges(), "?b2")
        .out("?node1", "?node2", "?is-new");

    iterate = Api.selectFields(iterate,
        new Fields("?node1", "?node2", "?is-new"));
    return (Subquery) iterate;
}

```

源 tap 从前面的迭代中生成 PersonID 的元组。

从声明的查询输出中，JCascalog 使用“b1”对元组分组。

移除分组标识符，因为不再需要它。

该子查询提出了算法的逻辑；完成迭代步骤需要添加合适的源 tap 和目的 tap，并执行查询：

```

public static Tap userIdNormalizationIteration(int i) {
    Tap source = (Tap) Api.hfsSeqfile("/tmp/swa/equivs" + (i - 1));
    Tap sink = (Tap) Api.hfsSeqfile("/tmp/swa/equivs" + i);
    Tap progressSink = (Tap) Api.hfsSeqfile("/tmp/swa/equivs" + "-new");

    Subquery iteration = iterationQuery(source);
    Subquery newEdgeSet = new Subquery("?node1", "?node2")
        .predicate(iteration, "?node1", "?node2", "_")
        .predicate(Option.DISTINCT, true);
    Subquery progressEdges = new Subquery("?node1", "?node2")
        .predicate(iteration, "?node1", "?node2", true);

    Api.execute(Arrays.asList(sink, progressSink),
        Arrays.asList(newEdgeSet, progressEdges));

    return progressEdgesSink;
}

```

所有边都生成到迭代步骤的输出。

新的边另外存储在独立的路径中。

避免将重复的边写入终端。

并行执行 newEdgeSet 和 progressEdges 查询。

只有该次迭代的新边才被写入 progress 终端。

除了将所有边存储为下一次迭代的输入，迭代步骤还在一个单独的文件夹中存储新的边。这为确定当前迭代是否生成了任何新的边提供了一种简单的方法；如果没有生成新的边，那么说明迭代已经达到了定点。下面的代码实现了迭代循环和终止逻辑：

```

追踪当前的
迭代。
public static int userIdNormalizationIterationLoop() {
    int iter = 1;
    while(true) {
        Tap progressEdgesSink = userIdNormalizationIteration(iter);
        FlowProcess flowProcess = new HadoopFlowProcess(new JobConf());
        if(!flowProcess.openTapForRead(progressEdgesSink).hasNext()) {
            return iter;
        }
        iter++;
    }
}

```

如果该次迭代期间没有生成新边，则终止迭代。

如果生成了新边，那么增加计数器并循环。

最后一次迭代决定了最终输出的路径。

下一个步骤是使用新的规范化的 PersonID 来更新页面浏览数据中的 PersonID。完成该步骤的管道图如图 9-5 所示，其中包括一个连接。

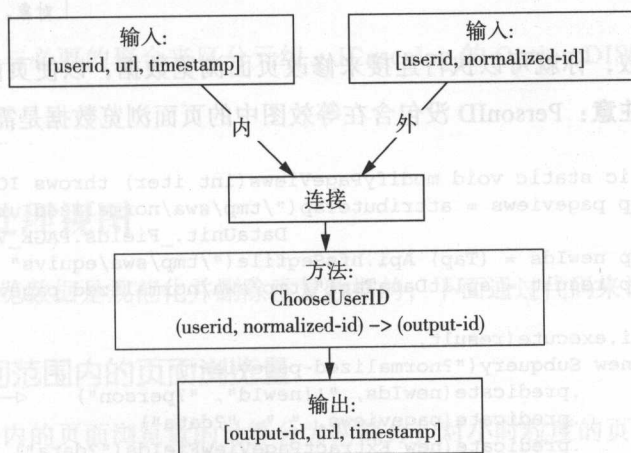


图 9-5 用户标识符规范化的最后步骤

在执行该连接之前，你需要两个自定义函数。首先，你必须分解 Thrift 页面浏览对象来提取必要的字段：

```

从 PageView-Edge 对象中
提取相关的
参数。
public static class ExtractPageViewFields extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        Data data = (Data) call.getArguments().getObject(0);
        PageViewEdge pageview = data.get_dataunit().get_page_view();
        if(pageview.get_page().getSetField() == PageID.Fields.URL) {

```


生成页面浏览数据的 URL、PersonID 和时间戳。

```
call.getOutputCollector().add(
    new Tuple(pageview.get_page().get_url(),
              pageview.get_person(),
              data.get_pedigree().get_true_as_of_secs()));
}
```

尽管时间戳不是立即需要的，但该函数将在 workflows 的其他部分被重用。

第二个所需的函数用于获取页面浏览的 Data 对象和新的 PersonID，并返回带有已更新 PersonID 的页面浏览的新 Data 对象：

备份 Data 对象，这样它就可以安全地被修改。

如果页面浏览的 PersonID 不是等效图的一部分，那么 newId 可能是空。

```
public static class MakeNormalizedPageview extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        PersonID newId = (PersonID) call.getArguments().getObject(0);
        Data data = ((Data) call.getArguments().getObject(1)).deepCopy();
        if(newId != null) {
            data.get_dataunit().get_page_view().set_person(newId);
        }
        call.getOutputCollector().add(new Tuple(data));
    }
}
```

生成可能已修改的页面浏览的 Data 对象。

有了这两个函数，你就可以执行连接来修改页面浏览数据，以使页面浏览数据使用规范化的 PersonID。注意：PersonID 没包含在等效图中的页面浏览数据是需要外连接的：

使用来自迭代循环的最终输出。

```
public static void modifyPageViews(int iter) throws IOException {
    Tap pageviews = attributeTap("/tmp/swa/normalized_urls",
                                DataUnit.Fields.PAGE_VIEW);
    Tap newIds = (Tap) Api.hfsSeqfile("/tmp/swa/equivs" + iter);
    Tap result = splitDataTap("/tmp/swa/normalized_pageview_users");
```

在页面浏览数据中的用户标识符上做连接。

```
Api.execute(result,
    new Subquery("?normalized-pageview")
        .predicate(newIds, "!!newId", "?person")
        .predicate(pageviews, "_", "?data")
        .predicate(new ExtractPageViewFields("?data")
            .out("_", "?person", "_")
            .predicate(new MakeNormalizedPageview(), "!!newId", "?data")
            .out("?normalized-pageview"));
}
```

在“?person”字段上执行连接；“!!newId”的前缀表明这是一个外连接。

创建并生成新的规范化的页面浏览数据。

最后一个任务是定义一个封装函数来执行 workflow 步骤的不同阶段：

```
public static void normalizeUserIds() throws IOException {
    initializeUserIdNormalization();
    int numIterations = userIdNormalizationIterationLoop();
    modifyPageViews(numIterations);
}
```

这总结了工作流的用户标识符规范化内容。这个示例很好地展示了使用通用编程语言的库指定 MapReduce 计算的优势。逻辑的重要组成部分，如迭代和定点检查，是用普通的 Java 代码编写的。你应该也注意到了，根据管道图得到的代码和前一章中列出的伪代码是很接近的。这表明你工作在正确的抽象层次上。

9.6 页面浏览去重

下一个步骤是在计算批处理视图的准备工作中删除重复的页面浏览。这段代码非常简单，我们将跳过管道图直接看代码：

```
public static void deduplicatePageviews() {
    Tap source = attributeTap("/tmp/swa/normalized_pageview_users",
        DataUnit.Fields.PAGE_VIEW);
    Tap outTap = splitDataTap("/tmp/swa/unique_pageviews");
    Api.execute(outTap,
        new Subquery("?data")
            .predicate(source, "?data")
            .predicate(Option.DISTINCT, true));
}
```

限制源 tap 只读取 Pail 中的页面浏览。

distinct 谓词会删除所有重复的页面浏览对象。

对于插入分组与必要的聚合来区分元组，JCascalog 的 Option.DISTINCT 谓词是很方便的。

9.7 计算批处理视图

现在的页面浏览数据是规范化并删除重复数据的，下面通过代码来计算批处理视图。

9.7.1 给定时间范围内的页面浏览量

给定时间范围内的页面浏览量的计算分为两部分：对小时粒度的页面浏览量进行计数；将每小时的计数相加得到所有所需的粒度。

第一部分的管道图如图 9-6 所示。

首先编写为时间戳确定小时桶的函数：

```
public static class ToHourBucket extends CascalogFunction {
    private static final int HOUR_IN_SECS = 60 * 60;

    public void operate(FlowProcess process, FunctionCall call) {
        int timestamp = call.getArguments().getInteger(0);
        int hourBucket = timestamp / HOUR_IN_SECS;
    }
}
```

```
call.getOutputCollector().add(new Tuple(hourBucket));
}
```

该函数是一个非常标准的 JCascalog 查询，用来确定每小时的计数：

```
public static Subquery hourlyRollup() {
    Tap source = new PailTap("/tmp/swa/unique_pageviews");
    return new Subquery("?url", "?hour-bucket", "?count")
        .predicate(source, "?pageview")
        .predicate(new ExtractPageViewFields(), "?pageview")
        .out("?url", "_", "?timestamp")
        .predicate(new ToHourBucket(), "?timestamp")
        .out("?hour-bucket")
        .predicate(new Count(), "?count");
}
```

重用前面的页面浏览提取代码。

根据“?url”和“?hour-bucket”分组。

同样，管道图和 JCascalog 代码之间的映射是非常直接的。

第二部分的管道图如图 9-7 所示。

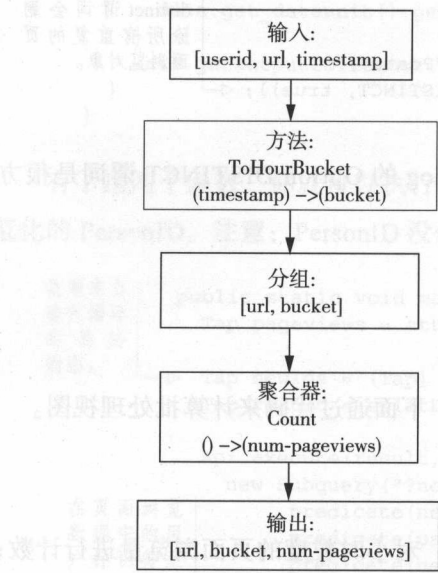


图 9-6 计算小时粒度的给定时间范围内的页面浏览量

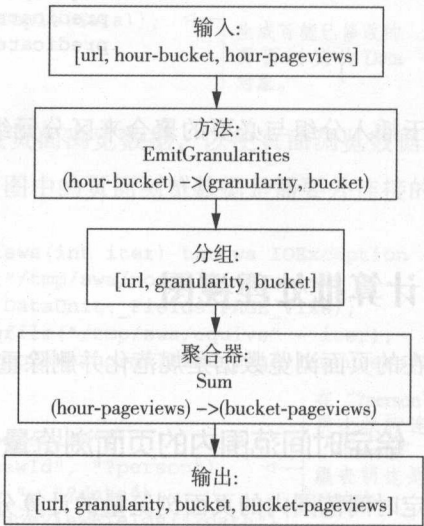


图 9-7 所有时间粒度的给定时间范围内的页面浏览量

下面先从一个函数开始，该函数用来生成给定小时桶的所有粒度：

```
public static class EmitGranularities extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        int hourBucket = call.getArguments().getInteger(0);
        int dayBucket = hourBucket / 24;
        int weekBucket = dayBucket / 7;
        int monthBucket = dayBucket / 28;
    }
}
```

该函数为每个输入生成 4 个二元组。

```

call.getOutputCollector().add(new Tuple("h", hourBucket));
call.getOutputCollector().add(new Tuple("d", dayBucket));
call.getOutputCollector().add(new Tuple("w", weekBucket));
call.getOutputCollector().add(new Tuple("m", monthBucket));
}
}

```

第一个元素用 h、d、w 或 m 来表明小时、天、周、月的粒度；第二个元素是时间桶的数值。

利用该函数，计算所有粒度的加和是很容易的：

```

public static Subquery pageviewBatchView() {
    Subquery pageviews =
        new Subquery("?url", "?granularity", "?bucket", "?total-pageviews")
            .predicate(hourlyRollup(), "?url", "?hour-bucket", "?count")
            .predicate(new EmitGranularities(), "?hour-bucket")
            .out("?granularity", "?bucket")
            .predicate(new Sum(), "?count").out("?total-pageviews");
    return pageviews;
}

```

生成所有粒度的桶。

执行每小时计数的子查询。

根据 url、粒度与桶，加和页面浏览量的计数。

9.7.2 给定时间范围内的独立访客

给定时间范围内的独立访客，与给定时间范围内的页面浏览量是类似的，但它不是计数，为此你需要创建 HyperLogLog 集。计算该批处理视图需要两个新的自定义操作。

首先是一个聚合器，用于从用户标识符的序列中构造一个 HyperLogLog 集：

```

public static class ConstructHyperLogLog extends CascalogBuffer {
    public void operate(FlowProcess process, BufferCall call) {
        HyperLogLog hll = new HyperLogLog(8192);
        Iterator<TupleEntry> it = call.getArgumentsIterator();
        while(it.hasNext()) {
            TupleEntry tuple = it.next();
            hll.offer(tuple.getObject(0));
        }
        try {
            call.getOutputCollector().add(new Tuple(hll.getBytes()));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}

```

该函数为访客集合生成一个 HyperLogLog 集。

使用 1 KB 的存储构造一个 HyperLogLog 集。

将所有对象添加到集合中。

生成 HyperLogLog 对象的存储字节。

下一个函数是另一个自定义聚合器，用于合并小时粒度的 HyperLogLog 集，以组成更大时间间隔的 HyperLogLog 集：


```

public static class MergeHyperLogLog extends CascalogBuffer {
    public void operate(FlowProcess process, BufferCall call) {
        Iterator<TupleEntry> it = call.getArgumentsIterator();
        HyperLogLog merged = null;
        try {
            while(it.hasNext()) {
                TupleEntry tuple = it.next();
                byte[] serialized = (byte[]) tuple.getObject(0);
                HyperLogLog hll = HyperLogLog.Builder.build(serialized);
                if(merged == null)
                    merged = hll;
                else {
                    merged = (HyperLogLog) merged.merge(hll);
                }
            }
            call.getOutputCollector().add(new Tuple(merged.getBytes()));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

```

创建一个新的 HyperLogLog 集来存储合并的结果。

将当前集合合并到结果中。

从存储字节中重构 HyperLog-Log 集。

为合并的集合生成存储字节。

下面的代码清单使用这些操作来计算批处理视图。注意其与给定时间范围内的页面浏览量查询的相似之处：

```

public static void uniquesView() {
    Tap source = new PailTap("/tmp/swa/unique_pageviews");

    Subquery hourlyUniques =
        new Subquery("?url", "?hour-bucket", "?hyper-log-log")
            .predicate(source, "?pageview")
            .predicate(new ExtractPageViewFields(), "?pageview")
            .out("?url", "?user", "?timestamp")
            .predicate(new ToHourBucket(), "?timestamp")
            .out("?hour-bucket")
            .predicate(new ConstructHyperLogLog(), "?user")
            .out("?hyper-log-log");

    Subquery uniques =
        new Subquery("?url", "?granularity", "?bucket", "?aggregate-hll")
            .predicate(hourlyUniques, "?url", "?hour-bucket", "?hourly-hll")
            .predicate(new EmitGranularities(), "?hour-bucket")
            .out("?granularity", "?bucket")
            .predicate(new MergeHyperLogLog(), "?hourly-hll")
            .out("?aggregate-hll");

    return uniques;
}

```

第一个子查询为每个 URL 确定了每小时的 HyperLog-Log 集。

第二个子查询确定了所有粒度的 HyperLogLog 集。

还可以创建一个函数，用来抽象页面浏览量查询和独立访客查询的通用部分。我们将把它作为练习留给读者。

进一步优化 HyperLogLog 批处理视图

所展示的实现为每个 HyperLogLog 集使用了相同的大小——1 KB。为了能够给可能收到数百万或数以亿计访问次数的 URL 一个合理准确的答案，HyperLogLog 集需要这么大的存储空间。但大多数使用 SuperWebAnalytics.com 的网站不会获得这么多的页面浏览量，所以为 HyperLogLog 集使用这么大的存储空间是很浪费的。

为了进一步优化，你可以查看该域名上 URL 的总页面浏览次数，并调整对应的 HyperLogLog 集的大小。使用这种方法可以大大减少批处理视图所需的空间，代价是为视图生成代码增添了一些复杂性。

9.7.3 跳出率分析

最后一个批处理视图是计算每个 URL 的跳出率。其管道图如图 9-8 所示。



图 9-8 跳出率分析的管道图

该批处理视图的关键是 AnalyzeVisits 聚合器，它用于查看一个用户在一个域名上所生成的所有页面浏览，并计算访问次数和反弹访问次数。最简单的计算方法是按照已排序的顺序查看页面浏览。若页面浏览之间的间隔超过 30min，则开始一次新的访问。如果访问只包含一个页面，那么它被认为是一次反弹。

下面的聚合器实现了这个逻辑。相关的查询将确保提供给该聚合器的输入是按已排序的顺序的：

```
public static class AnalyzeVisits extends CascalogBuffer {
    private static final int VISIT_LENGTH_SECS = 60 * 30;

    public void operate(FlowProcess process, BufferCall call) {
        Iterator<TupleEntry> it = call.getArgumentsIterator();
        int bounces = 0;
        int visits = 0;
        Integer lastTime = null;
        int numInCurrVisit = 0;

        while(it.hasNext()) {
            TupleEntry tuple = it.next();
            int timeSecs = tuple.getInteger(0);
            if(lastTime == null || (timeSecs - lastTime) > VISIT_LENGTH_SECS) {
                visits++;
                if(numInCurrVisit == 1) {
                    bounces++;
                }
                numInCurrVisit = 0;
            }
            numInCurrVisit++;
        }
        if(numInCurrVisit == 1) {
            bounces++;
        }
        call.getOutputCollector().add(new Tuple(visits, bounces));
    }
}
```

如果两个连续的页面浏览间隔少于 30min，那么它们属于同一次访问。

假设页面浏览是按时间顺序排序的。

追踪前一次页面浏览的时间。

注册新一次访问的开始。

确定前一次访问是否是反弹访问。

确定上一次页面浏览是否是反弹访问。

生成访问计数和反弹访问计数。

在实现子查询之前，先来实现下一个它所需要的自定义函数。该函数用于从 URL 中提取域名：

```
public static class ExtractDomain extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        String urlStr = call.getArguments().getString(0);
        try {
            URL url = new URL(urlStr);
            call.getOutputCollector().add(new Tuple(url.getAuthority()));
        } catch (MalformedURLException e) {}
    }
}
```

使用 Java 本地库来提取域名。

下面把所有代码组合在一起，生成跳出率分析的计算：

```
public static Subquery bouncesView() {
    Tap source = new PailTap("/tmp/swa/unique_pageviews");

    Subquery userVisits =
        new Subquery("?domain", "?user", "?num-user-visits",
                    "?num-user-bounces")
            .predicate(source, "?pageview")
            .predicate(new ExtractPageViewFields(), "?pageview")
            .out("?url", "?user", "?timestamp")
            .predicate(new ExtractDomain(), "?url")
            .out("?domain")
            .predicate(Option.SORT, "?timestamp")
            .predicate(new AnalyzeVisits(), "?timestamp")
            .out("?num-user-visits", "?num-user-bounces");

    Subquery bounces =
        new Subquery("?domain", "?num-visits", "?num-bounces")
            .predicate(userVisits, "?domain", "_",
                    "?num-user-visits", "?num-user-bounces")
            .predicate(new Sum(), "?num-user-visits")
            .out("?num-visits")
            .predicate(new Sum(), "?num-user-bounces")
            .out("?num-bounces");

    return bounces;
}
```

将页面浏览按时间顺序排序来分析访问次数——Option.SORT谓词允许你控制每一组在进入聚合器操作之前是如何排序的。

确定每个用户的反弹次数和访问次数。

加和所有用户的反弹次数和访问次数，以计算批处理视图。

放松，做个深呼吸！经过这么长时间，付出了这么多努力，你已经成功地完成了 SuperWebAnalytics.com 基于重新计算的层。

9.8 总结

SuperWebAnalytics.com 的批处理层只有几百行代码，但是涉及的业务逻辑很复杂。各种抽象很好地结合在一起——想要在每个步骤中实现什么和如何实现之间是很直接的映射。由于工具集的特性，总是会出现一些晦涩难懂的细节——尤其是 Hadoop 的小文件问题——但这些并不难克服。

虽然本章已经相当深入地探讨了具体工具的细节，但回顾并记住批处理层的首要原理是很重要的。不变性和重新计算的概念提供了容忍人为错误这个属性，这是任何数据系统都不可缺少的属性。你知道了编写代码生成批处理视图是多么得简单——计算框架已经为你处理了困难的东西，如容错性和并发性。批处理层极大地简化了生成实时视图的问题，因为实时视图只需要计算完整数据集很少的一部分。以后，当你了解实时计算固有的复杂性时，你会感激批处理层宽松的延迟要求——这也使得批处理层的各方面操作起来更简单。

接下来将继续探讨服务层，这样批处理视图就能够以随机访问的方式被快速读取。

□ Lambda 架构中服务层的需求

□ 服务层如何解决长期争论的规范化和非规范化问题

下面从讨论构建服务层视图时面临的关键问题开始。

第二部分 Part 2

服务层

第二部分主要介绍 Lambda 架构的服务层。服务层包含索引和服务批处理层结果的数据库。第二部分较为简短，因为不需要随机写的数据库是非常简单的。第 10 章讨论了服务层的高层概念，而第 11 章展示了一个名为“ElephantDB”的服务层数据库示例。

服务层概述

本章内容

- ❑ 根据服务的查询定制批处理视图
- ❑ 数据规范化和非规范化争论的一个新答案
- ❑ 批量写入、随机读取和不随机写入的数据库的优点
- ❑ 对比 Lambda 架构解决方案与全增量解决方案

此时，你已经了解了如何利用批量计算来预先计算任何数据集的任意视图。因为视图是有用的，所以你必须能够低延迟地访问其内容，如图 10-1 所示，这就是服务层的作用。服务层索引视图，并提供接口，以便预先计算的数据可以被快速查询到。

服务层是 Lambda 架构批处理部分的最后一个组件。它与批处理层紧密相关，因为批处理层负责持续更新服务层视图。由于批量计算的高延迟特性，这些视图将总是过时的。但这不是问题，因为速度层将负责处理任何在服务层还不可用的数据。

不幸的是，服务层是一个“工具落后于理论”的领域。构建一个通用的服务层实现不会很难——实际上，它比构建任何目前现有的 NoSQL 数据库都更容易。本章将介绍创建一个简单、可扩展、容错和通用的服务层背后的完整理论，然后使用可利用的最佳工具来说明底层概念。

通过讨论服务层的相关知识点，你将了解以下内容：

- ❑ 最小化延迟、资源使用和方差的索引策略

□ Lambda 架构中服务层的需求

□ 服务层如何解决长期争论的规范化和非规范化问题

下面从讨论构建服务层视图时面临的关键问题开始。

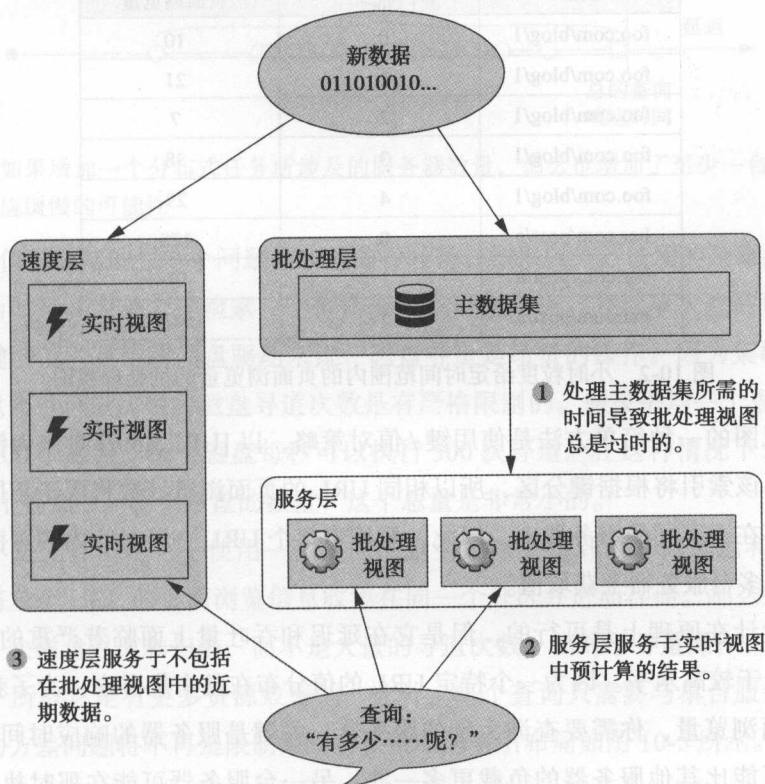


图 10-1 在 Lambda 架构中，在主数据集上执行计算的结果时，服务层提供低延迟访问。

由于批量计算需要时间，服务层视图稍微有些过时

10.1 服务层的性能指标

与批处理层一样，由于可扩展性，服务层分布在多台机器上。服务层的索引以完全分布式的方式被创建、加载和服务。

当设计这些索引时，你必须考虑两个主要的性能指标：延迟和吞吐量。在这种背景下，延迟是响应单个查询所需的时间，而吞吐量是给定时间内可以服务的查询数量。服务层索引的结构与这些指标之间的关系，最好通过例子进行解释。

下面将简要回顾一下长期运行的 SuperWebAnalytics.com 示例——尤其是给定时间范围内的页面浏览量查询。其目标是服务特定 URL 的每个小时页面浏览量和特定时间范围内的

页面浏览量。为了进一步简化该讨论,假设页面浏览计数只使用小时粒度来生成。生成的视图类似于图 10-2 所示的样式。

URL	桶	页面浏览量
foo.com/blog/1	0	10
foo.com/blog/1	1	21
foo.com/blog/1	2	7
foo.com/blog/1	3	38
foo.com/blog/1	4	29
bar.com/post/a	0	178
bar.com/post/a	1	91
bar.com/post/a	2	568

图 10-2 小时粒度给定时间范围内的页面浏览量的批处理视图

索引该视图的一种简单方法是使用键/值对策略,以 [URL, hour] 对作为键,以页面浏览量作为值。该索引将根据键分区,所以相同 URL 的页面浏览计数将属于不同的分区。不同的分区将存在于不同的服务器上,因此,要检索单个 URL 一段时间内的页面浏览量,需从服务层的多台服务器上获取值。

虽然该设计在原理上是可行的,但是它在延迟和吞吐量上面临着严重的问题。首先,延迟会持续处于较高水平。因为一个特定 URL 的值分布在整个集群中,为了获得很长时间范围内的页面浏览量,你需要查询大量的服务器。关键是服务器的响应时间不同。例如,一台服务器可能比其他服务器的负载更多一些;另一台服务器可能在那时执行垃圾回收。即使能够并行化地获取请求,总体查询响应时间也会受到速度最慢的服务器的限制。

为了说明这一点,假设一个查询需要从 3 台服务器上获取数据。响应时间的分布的代表性样本如图 10-3 所示。

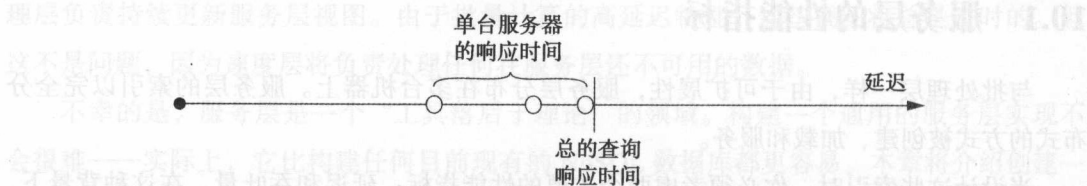


图 10-3 当一个任务被分布在多台服务器上时,总体延迟取决于最慢的服务器的响应时间

相比之下,假设该查询命中了 20 台服务器。典型的延迟分布如图 10-4 所示。

通常来说,一个查询涉及的服务器越多,那么查询的总延迟就越高。这缘于一个简单的事实——涉及更多的服务器,将增加至少一台服务器响应缓慢的可能性。因此,服务器

响应时间的不同导致一台服务器的最差性能变成查询的普遍性能。这是为给定时间范围内的页面浏览量查询实现良好延迟所面临的一个严重问题。

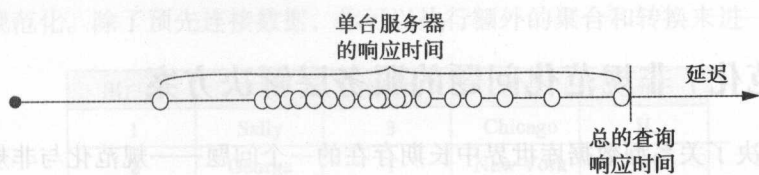


图 10-4 如果增加一个分布式任务所涉及的服务器数量，那么也增加了至少一台服务器响应缓慢的可能性

使用键/值对策略的另一个问题是较差的吞吐量，如果服务器使用的是磁盘而不是固态硬盘，则尤为如是。为单个键检索一个值需要一次磁盘寻道，并且单次查询可能要获取几十个或更多键的值。对传统硬盘驱动来说，磁盘寻道是昂贵的操作。因为集群中磁盘的数量有限，所以每秒可以实现的磁盘寻道次数是有严格限制的。假设平均一个查询获取 20 个键，集群有 100 个磁盘，每个磁盘每秒可以执行 500 次寻道。在这种情况下，集群每秒只能服务 2500 个查询——鉴于磁盘的数量，这个总量是非常小的。

但你并不是无计可施——使用一种不同的索引策略会得到更好的延迟和吞吐量特性。这个想法是将单个 URL 的页面浏览信息收集在同一个分区并进而存储。那么获取页面浏览量将只需要一次磁盘寻道和扫描，而不是大量的寻道次数。相对于寻道操作，扫描是很“便宜”的操作，所以它是有更多资源效率的。此外，每个查询只需要与单台服务器通信，所以前面策略的方差问题将不再是限制因素。该策略的索引布局如图 10-5 所示。

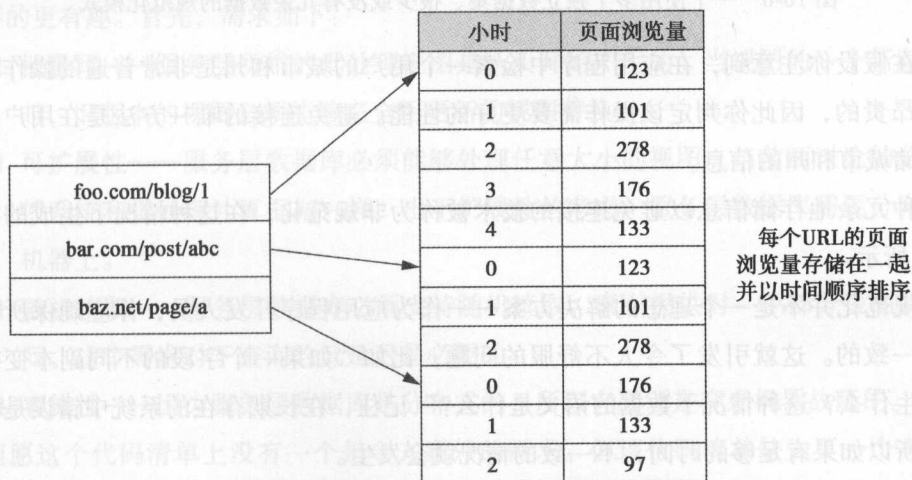


图 10-5 排序的索引促进扫描并限制磁盘寻道，从而改善了延迟和吞吐量

这两个示例证明了构建服务层索引的方式对查询的性能有显著影响。Lambda 架构一个至关重要的优势是，它允许根据查询定制服务层来提升效率。

10.2 规范化 / 非规范化问题的服务层解决方案

服务层解决了关系型数据库世界中长期存在的一个问题——规范化与非规范化的两难境地。为了解决方案及其影响，首先你需要了解潜在的问题。

在所有不可忽略的权衡之间，最后需要在规范化和非规范化之间做出选择。在关系型的世界里，你想要存储完全规范化的数据，其中包括在独立数据集之间定义关联以减少冗余。不幸的是，查询规范化的数据可能会很慢，所以可能需要存储一些冗余信息来改善响应时间。非规范化过程提高了性能，但它在保持冗余数据一致性方面有巨大的复杂性。

为了阐释这种紧张关系，假设你在关系型表中存储用户的位置信息（见图 10-6），每个位置都有一个标识符，每个人使用这些标识符的其中一个来表示他们的位置，那么一个为特定个体检索位置的查询就需要两个表之间的连接。这是完全规范化模式的一个示例，因为没有信息被冗余存储。

用户ID	名称	位置ID	位置ID	城市	州	人口
1	Sally	3	1	New York	NY	8.2M
2	George	1	2	San Diego	CA	1.3M
3	Bob	3	3	Chicago	IL	2.7M

图 10-6 一个使用多个独立数据集、很少或没有冗余数据的规范化模式

现在假设你注意到，在应用程序中检索一个用户的城市和州是非常普遍的操作。由于连接是昂贵的，因此你判定该操作需要更好的性能。避免连接的唯一方法是在用户表中冗余地存储城市和州的信息。

这种冗余地存储信息以避免连接的技术被称为非规范化，在这种情况下生成的模式如图 10-7 所示。

非规范化并不是一个理想的解决方案——作为应用程序开发人员，你应确保所有冗余数据是一致的。这就引发了令人不舒服的问题，比如“如果一个字段的不同副本变得不一致会发生什么？这种情况下数据的语义是什么？”记住，在长期存在的系统中错误是不可避免的，所以如果有足够的时间，不一致的情况就会发生。

幸运的是，Lambda 架构中主数据集和服务层之间的分离，解决了规范化与非规范化的问题。在批处理层中，你可以随心所欲地规范化主数据集。批处理层上的计算批量地读取主数

数据集，所以没有必要设计模式来为随机访问读取做优化。作为互补，服务层是完全根据它服务的查询来定制的，所以可以根据需要进行优化来获得最大的性能。服务层的这些优化可以远远超过非规范化。除了预先连接数据，你可以执行额外的聚合和转换来进一步提升效率。

用户ID	名称	位置ID	城市	州
1	Sally	3	Chicago	IL
2	George	1	New York	NY
3	Bob	3	Chicago	IL

位置ID	城市	州	人口
1	New York	NY	8.2M
2	San Diego	CA	1.3M
3	Chicago	IL	2.7M

图 10-7 使用非规范化表冗余地存储数据以提升查询性能

关于 Lambda 架构中一致性的问题，在批处理层和服务层之间冗余地存储信息是完全正确的。关键的区别是，服务层被定义成主数据集的一个函数。如果一个错误引发了不一致问题，那么可以通过从头开始重新计算服务层，很容易地纠正它们。

10.3 服务层数据库的需求

Lambda 架构对服务层数据库提出了一些特定的需求。但服务层数据库不需要的需求远比需要的更有趣。首先，需求如下：

- ❑ **批量写**——服务层所用的批处理视图是从头开始产生的。当视图的一个新版本可用时，旧版本的视图必须能够完全用更新的视图替换。
- ❑ **可扩展性**——服务层数据库必须能够处理任意大小的视图。与前面讨论的分布式文件系统和批处理计算框架一样，出于扩展性的考虑，服务层数据库需要分布在多台机器上。
- ❑ **随机读取**——服务层数据库必须支持随机读取，索引提供对一小部分视图的直接访问。这个需求对于查询的低延迟是必要的。
- ❑ **容错性**——因为服务层数据库是分布式的，所以它必须是容忍机器故障的。

但愿这个代码清单上没有一个是令人意外的需求。但该代码清单缺失了一个习惯性的需求——所有熟悉的数据库上的一个标准需求——**随机写**。该功能与服务层完全无关，因为视图只是批量生产的。需要明确的是，Lambda 架构中确实存在随机写，但是它们被隔离

在速度层中以实现低延迟更新。更新服务层将产生全部的新视图，所以服务层数据库不需要具备修改部分当前视图的能力。

这是一个令人惊奇的结果，因为随机写是导致数据库中大部分复杂性的原因——在分布式数据库中甚至更复杂。试想一下，例如，第1章中讨论的随机写数据库是如何工作的一个讨厌的细节——需要通过压缩来回收未使用的空间。作为一个密集型的操作，不定期地压缩会占据机器的许多资源。如果没有正确地管理压缩，机器将会超载，并且当负载转移到其他机器上时，可能会变成级联的失败。

因为服务层不需要随机写，它不需要在线压缩，所以这种复杂性及其相关的操作负担，在服务层中完全消失。在考虑服务层和速度层集群的相对大小时，这是极其重要的。服务层生成了绝大部分主数据集的视图（可能超过99%），所以它需要大部分的数据库资源。这意味着绝大多数的数据库服务器不受管理在线压缩的操作负担之苦。

当数据库必须支持随机写时，在线压缩只是它引入的大量复杂性之一。另一个复杂性是需要同步读操作和写操作，这样写到一半的值就不会被读取。当数据库没有随机写时，它可以优化读取路径，并比随机读/写数据库获得更好的性能。

一个粗略但良好的复杂性指标可以从代码库的大小看出。ElephantDB，专门用来构建服务层数据库，只有几千行代码。HBase和Cassandra，两个流行的分布式读/写数据库，有几十万行代码。代码行的数量通常不是一个好的复杂性指标，但在这种情况下应该可以看出它们惊人的差异。

一个更简单的数据库更容易预测，因为它做更少的事情。因此不太可能有错误——就像你看到的压缩——很大程度上将更容易操作。因为服务层视图包含绝大多数可查询的数据，服务层的基本简单性对于总体架构的鲁棒性是一个巨大的福音。

10.4 设计 SuperWebAnalytics.com 的服务层

现在回到 SuperWebAnalytics.com 的示例，并为其设计理想的服务层。之前我们已经为 Super WebAnalytics.com 构建了一个批处理工作流，并为3个查询生成了批处理视图：给定时间范围内的页面浏览量、给定时间范围内的独立访客和跳出率分析。批处理层的输出是没有索引的——这是服务层的工作，索引这些视图并以低延迟地服务于它们。

本书专注于 SuperWebAnalytics.com 的理想的服务层设计。比起 Lambda 架构中其他的任何组件，服务层都是实际工具是要落后于理想工具的。这颇具讽刺意味，因为服务层数据库是 Lambda 架构所需的工具中最简单且最容易构建的。我们相信这归结于历史的推

动——大部分人构建的应用程序由单个庞大的数据库集群服务，而数据库集群被更新为使用实时、增量的更新操作。为了给未来的工具提供发展蓝图，预测什么是可能理想的工具是非常重要的。事实上，你可能会发现为服务层重用了传统数据库。

下面来看 SuperWebAnalytics.com 每个视图理想的索引类型。

10.4.1 给定时间范围内的页面浏览量

给定时间范围内的页面浏览量查询，检索一段时间范围内一个 URL 的页面浏览计数并把它们加和在一起。正如已经讨论过的，该查询的理想索引是键排序映射，如图 10-5 所示。

给定时间范围内的页面浏览量批处理视图，不仅仅计算了每小时粒度的分桶计数，还有每日、每周、每月和每年粒度的计数。这样做是为了减少解决一个查询必须被检索的值的总数——一年的范围需要检索数千个小时桶，但当使用更大的粒度时，只要检索几个桶。事实证明，如果使用键排序映射 (key-to-sorted-map) 的索引类型，这些更高的粒度就不再需要了。这是因为当一段时间范围内所有值按顺序存储时，一次性读取这些值是非常廉价的。

例如，假设排序映射中的每一项，从桶到页面浏览计数的映射，需要 12B (4B 用于桶的数量，8B 用于值)。检索两年期限的桶的计数需要大约 17 500 个值。如果把这些都加起来，那么相当于必须检索 205 KB 的总量。该总量很小，但最好能优化这些东西，使得即使整体需要读取更多的信息，也只需要一次寻道。

当然，这种分析是针对现在的硬盘的特点的，对于固态硬盘或其他工具，可能会得出不同的结论：一个包括更多粒度的索引将更优越。

10.4.2 给定时间范围内的独立访客

下面讨论给定时间范围内的独立访客的理想索引 (见图 10-8)。给定时间范围内的独立访客查询与给定时间范围内的页面浏览量非常类似——基于一段时间范围的值检索一个组合的值。不过一个显著的区别是，给定时间范围内的独立访客使用的 HyperLogLog 集，比给定时间范围内的页面浏览量的存储值的桶要大得多。所以，如果只有包含小时粒度的排序索引，且 HyperLogLog 集的大小是 1024B，那么对于一个两年期限的查询，你必须检索大约 17 MB 的 HyperLogLog 信息。如果硬盘可以支持 300 MB/s 的读吞吐量，那么只读取这些信息就需要 60 ms (并且假设这是完全理想的环境)。除此之外，合并 HyperLogLog 集比简单地求数字和更昂贵，可能会给查询添加更多的延迟。因为给定时间范围内的独立访

客，本质上比给定时间范围内的页面浏览量更昂贵，所以似乎利用更大的粒度会更好一些。

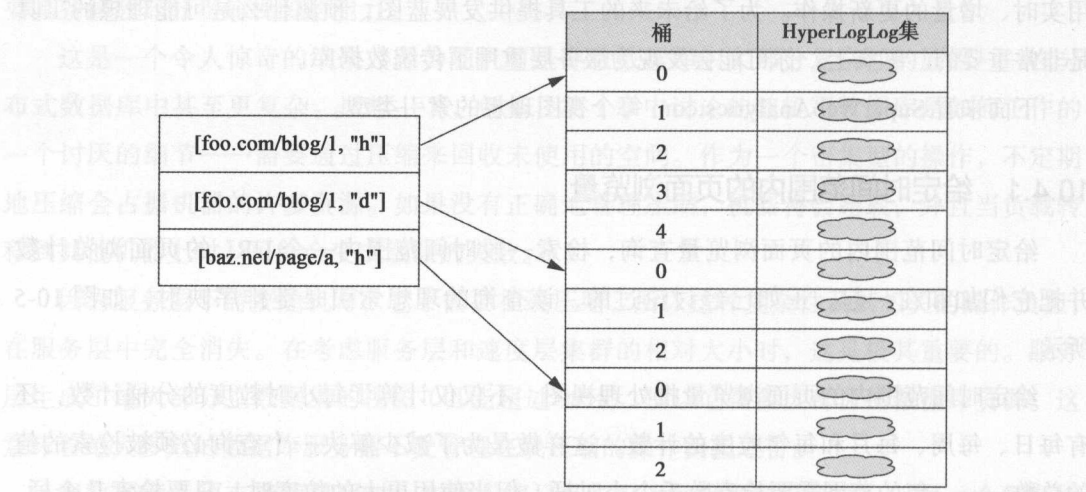


图 10-8 给定时间范围内的独立访客的索引设计。虽然索引的键是 URL 和粒度的组合，但只根据 URL 在服务器之间对索引进行分区

在这种情况下，图 10-8 中的索引似乎是最优的。该索引与给定时间范围内的页面浏览量使用的键到排序映射的索引相同，但有两个差异：

- ❑ 键是 URL 和粒度的组合。
- ❑ 索引只根据 URL 分区，而不是 URL 和粒度。为了检索一个 URL 和粒度的一系列值，可以使用 URL 找到包含所需要的信息的服务器，然后使用 URL 和粒度来查找所感兴趣的值。只根据 URL 分区能确保一个 URL 的所有桶被收集在相同的服务器上，并避免单个查询必须与多台服务器交互的任何方差问题。

10.4.3 跳出率分析

跳出率分析视图是从一个域名到该域名访问次数和反弹次数的映射。这是所支持的最简单的视图，因为它只需要一个键 / 值索引，如图 10-9 所示。

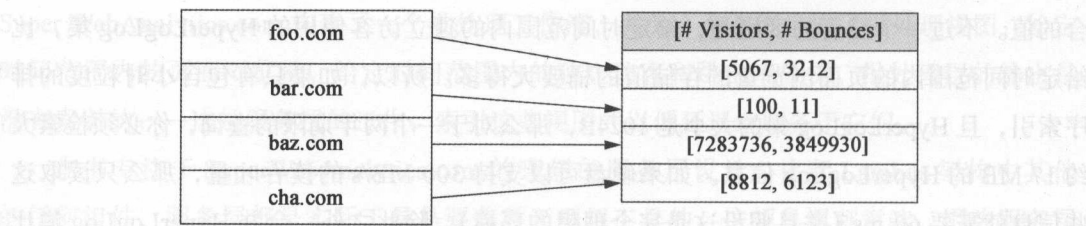


图 10-9 使用一个键 / 值对索引实现跳出率分析视图

10.5 对比全增量的解决方案

在前几章中，你已经了解了如何构建 SuperWebAnalytics.com 的批处理层和服务层。使用将视图计算为所有数据上的函数的这种模型，一切都相当简单。为了更好地了解这样一个系统所展示出的优秀属性，我们有必要将 Lambda 架构解决方案与使用全增量计算的传统架构进行对比。传统架构为了状态属性而使用大型读/写数据库，并且随着新数据的到达，要对该状态进行维护。

在第 1 章介绍 Lambda 架构时，我们对比了给定时间范围内的独立访客问题的传统解决方案与 Lambda 架构解决方案。我们已经介绍了所需的概念，现在来看它们细节上的差别。这里将展示给定时间范围内的独立访客问题的最著名的全增量的解决方案，你会发现最终的解决方案实现起来更复杂，明显没有那么准确，且有更糟糕的延迟和吞吐量特性，甚至需要特殊的硬件才可行。

10.5.1 给定时间范围内的独立访客的全增量方案

我们将逐步建立最好的全增量方案。为了开始这一过程，我们通过完全忽视最初解决方案中的等效边，来使这个问题更加简化。这将给带有等效边的独立访客这一更难的问题提供有价值的见解。

注意：在解决这一问题时，我们不会限制当前可用的工具。现有工具的任何合理变化都是允许的。我们感兴趣的是什么从根本上是可能的——使用最好的全增量解决方案是好，还是坏呢？

因为它是全增量解决方案，解决这个问题的关键是确定使用什么样的数据库和如何维护数据库中的状态。对于第一次尝试，让我们试着用键到集合（Key-to-set）的数据库。也就是说，该数据库实现了如下的一个接口：

```
interface KeyToSetDatabase {  
    Set getSet(Object key);  
    void addToSet(Object key, Object val);  
}
```

这样的数据库可以很容易地存在，并可以是分布式和容错的。使用这样的数据库而不是键/值数据库的原因是，它使 addToSet 操作更有效率。对于键/值数据库，你必须获取整个集合，添加元素，然后将整个集合写回。通过让数据库固有意地意识到所存储的数据结构，只需将所要添加的元素发送到集合中，这样的操作将更为高效。

任何全增量方法都有两个方面：确定当收到一个新的页面浏览时会发生什么（写方面）；

确定为了解决一个查询要计算什么(读方面)。对于写方面,数据库的键将被设置为 [URL, hour bucket] 对,值将被设置为该小时桶内访问该 URL 的所有 UserID 的集合。每当收到一个新页面浏览,UserID 将被添加到数据库适当的桶内。对于读方面,通过获取查询的时间范围内的所有桶,将集合合并在一起,然后通过计算该合并之后集合的独立访客计数来完成查询。

虽然很简单,但这种方法存在很多问题:

- ❑ 数据库占用非常大的空间,因为实际上每个页面浏览都要被存储在数据库中。
- ❑ 对于大范围时间的查询,你必须做多次的数据库查找操作。例如,一年的时间范围包含 8760 个桶。而获取 8760 个桶不利于快速查询。
- ❑ 对于受欢迎的网站,甚至个别桶中可能有数以百万计的元素(或更多)。这种情况也是非常不利于快速查询的。

下面采用一种不同的方法来大幅减少查询期间需要做的工作总量。对于第二种方法,我们利用 HyperLogLog 来近似集合并计数,并大大减少了所需的存储量。在这种尝试中,将使用 key-to-HyperLogLog 的数据库。此外,这样一个数据库没有理由不以分布式和容错性的形式存在——对于 Apache Cassandra 这样的数据库,这实际上是一个微小的变化。

和之前一样,键是 [URL, hour bucket] 对,值是代表该小时内访问该 URL 的所有 UserID 的 HyperLogLog 集。写方面只是将 UserID 添加到 HyperLogLog 集适当的桶中,读方面获取那段时间范围的所有 HyperLogLog 集,将这些 HyperLogLog 集合合并在一起,并得到计数。

因为 HyperLogLog 节省了巨大的空间,所有关于该方法的操作都会更有效率。现在个别桶已确保是很小的,且作为一个整体的数据库可以显著地高效利用空间。这是通过对查询精度做出非常温和的折中来实现的。

但是对于大时间范围的查询,这种方法仍有问题,它需要过多的数据库查找,而你想要大范围的查询与小范围的查询一样快速。

幸运的是,解决这个问题相当容易。对此,上一种方法再次使用了 key-to-HyperLogLog 的数据库,但是现在将键改变为 [URL, hour bucket, granularity] 的三联体。该想法的意思是:不是只计算小时粒度的 HyperLogLog 集,而是以更粗的粒度来计算它们,如天、周、月和年。

在写方面,每当传入一个新的页面浏览,UserID 将被添加到 HyperLogLog 集中适当的小时、天、周、月和年的桶。在读方面,通过读取最小数量的桶来计算结果。例如,对于从 2013 年 12 月 1 日到 2015 年 2 月 4 日的一个查询,只需要下面的桶:

❑ 2013 年 12 月的月

❑ 2014 年的年

❑ 2015 年 1 月的月

❑ 2015 年 2 月 1~3 日的日

相对于前面尝试的大范围查询所需读取的数以千计的桶，这是一个巨大的进步。该策略与 SuperWebAnalytics.com 批处理层视图中采取的方法几乎相同。正如你已经看到的，额外粒度的存储成本是最小的，所以为了使所有查询运行得快，略微增加一些存储是值得的。

总的来说，这是该问题非常令人满意的解决方法：对于所有查询是快速的，有效利用空间，易于理解，并且易于实现。现在重新将等效边引入这个问题中，看看一切都变成了什么样。在全增量架构中解决这个问题更加困难，而且你会发现最终的解决方案不是令人满意的。

如前所述，处理等效边问题的棘手之处在于，一个新的等效边可以改变任何可能查询的结果。例如，假设回到第一次尝试，每个 [URL, hour bucket] 对存储一组 UserID。假设你只打算在整个数据库中为每个人存储一个 UserID，所以每当传入一个新的等效边，你必须确保那个人只有一个 UserID 存在于整个数据库中。图 10-10 展示了这样一个数据库的示例，假设 UserID A 和 C 之间传入一个新的等效边。在这个例子中，需要修改数据库中所显示的 75% 的桶！因为你不知道哪些桶可能会受到影响，所以该方法中处理等效边的唯一方式是，遍历整个数据库中的每个等效边。这显然是不合理的。

Key (URL, Hour bucket)	Set of UserIDs
"foo.com/page1", 0	A, B, C
"foo.com/page1", 1	A, D
"foo.com/page1", 2	A, C, F
"foo.com/page1", 102	A, B, C, G

图 10-10 等效边可能影响数据库中的任何桶

试图优化该方法的一种可能方式是，维护从 UserID 到 UserID 所在的所有桶的集合的第二索引。如果一个用户只访问过两个桶，那么当传入等效边时，你只需要修改这两个桶中的 UserID，而不必遍历整个数据库。

不幸的是，这种方法存在很多问题。如果一个搜索引擎机器人每小时访问每个 URL 将会怎样？——该 UserID 的桶代码清单将包含数据库中的每一个桶，这是非常不切实际的。由于个别 UserID 拥有大量的桶列表或偶尔需要遍历大面积的数据库，这将使得许多合理的数据集性能不稳定。除了性能问题，还有复杂性问题。UserID 所属桶的信息存储在多个位置，这为数据库变得不一致开启了便利之门。

当处理等效边时，无法使用 HyperLogLog 应该也是显而易见的。HyperLogLog 集不知道它其中的元素，这使得它不可能应用等效边来去除冗余的 UserID。这是一个糟糕的结果，因为 HyperLogLog 曾是非常显著的优化。

到目前为止，我们已经概述了等效边的分析，它为每个人选出一个 UserID 作为表示。这个问题本身是很棘手的，并且增量地实现是相当复杂的。但由于该算法不需要理解全增量结构的复杂性，所以我们就假设这个问题已经被完全解决。该解决方案的结果是从 UserID 到 PersonID 的索引，PersonID 是选出来表示属于同一个人的所有 UserID 的标识符。

到目前为止，解决这个问题的难点在于，试图通过“修改”数据库在写端处理等效边，以确保等效边连接的两个 UserID 不同时存在于数据库中。所以不妨采取一种不同的方法——将处理等效边的工作移动到查询的读端。

在第一次读端的尝试中，如图 10-11 所示，数据库将会是键到集合的数据库——从 [URL, hour bucket] 到这一时段内访问该 URL 的所有 UserID 的集合。这一次，允许同一个人的多个 UserID 存在于数据库中，因为将在读取时处理等效边。读取操作如下：

- 1) 首先，获取给定时间范围内每个小时的每个 UserID 集，并将它们合并。
- 2) 使用 UserID-to-PersonID 索引将 UserID 集转换成 PersonID 集。
- 3) 返回 PersonID 集的计数。

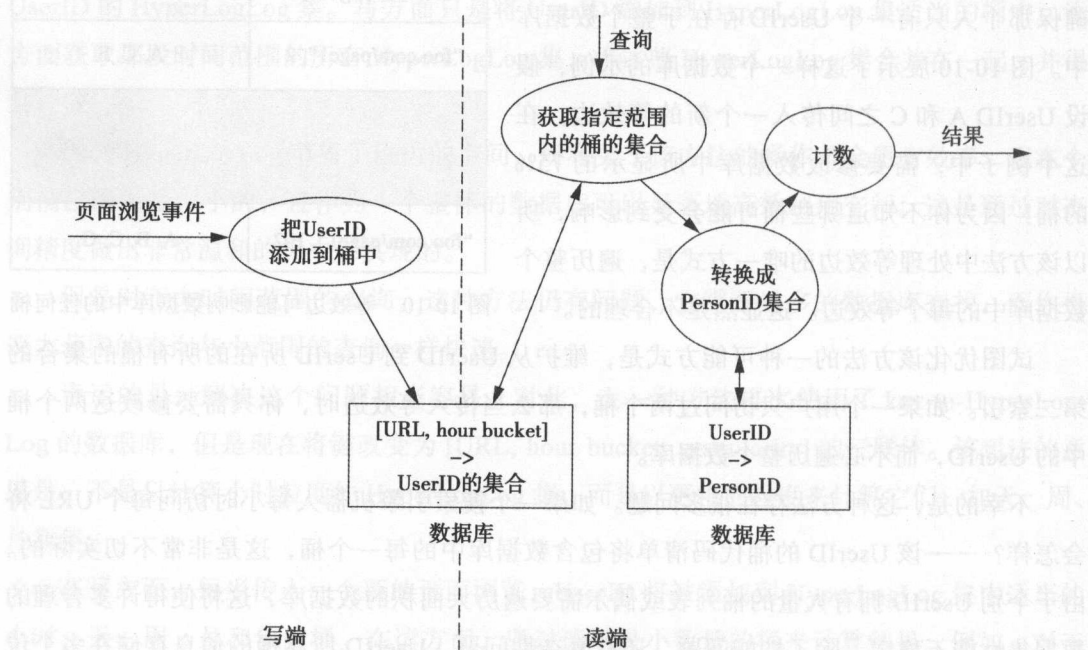


图 10-11 在工作流的读端处理等效边

不幸的是，这种方法不太可行，因为它太昂贵了。试想，一个查询有1亿个独立访客，这意味着你必须先通过获取许多GB的信息来得到UserID集，然后在UserID-to-PersonID索引中做1亿次查找。这项工作无法在几毫秒内完成。

对于前面的方法，可以通过使用近似法稍微进行修改，使之成为可行的方法，从而大大地减少存储和计算成本。这种方法的主题思想是，你不再是为每个桶存储整个UserID集，而是为每个桶存储UserID的样本。如果你只存储5%的UserID，获取UserID集时将减少95%的工作，并且UserID转换成PersonID时将最多减少95%的工作。通过将样本PersonID集的计数除以采样率，你会得到非样本集的计数的估计值。

抽样方法的写端和读端的工作流如图10-12所示。首次试抽样可能生成0~1的随机数，只有数字小于采样率的UserID才被添加到桶中。不幸的是，这并不能正常工作，这可以从一个简单的例子看出。假设有来自用户A、B、C、D的各100个页面浏览，所需的采样率为50%。因为每个用户有100个页面浏览，你几乎肯定会对所有的4个用户取样。这是错误的，因为适当的取样技术应该只取样平均的两个用户。

一种称为**散列取样**的不同技术能正确地取样。这种技术不是选择一个随机数来确定是否将UserID添加到桶中，而是使用像SHA-256这样的散列函数来散列UserID。散列函数有将输入均匀分布到输出数字的范围的属性。此外，散列函数是具有确定性的，所以相同的输入总是散列到相同的输出。有了这两个属性，如果只想取样25%的UserID，那么只需保留UserID的散列小于散列函数的最大输出值25%的所有UserID。由于散列函数的确定性，一个UserID一旦被取样，它将总是被取样；如果一个UserID没有被取样，它将永远不会被取样。所以50%的采样率意味着，无论每个UserID出现了多少次，你将总是保存集合的一半值。你可以使用散列取样来大大减少为每个桶存储的集合大小，而且所选择的采样率越高，查询的结果越准确。

好消息是，我们终于有一种可行的方法来实现高性能的查询。坏消息是，它有一些要注意的地方。

首先，散列取样方法的准确性级别与HyperLogLog是不尽相同的。对于和HyperLogLog一样的使用空间，平均误差将至少是原来的3~5倍，这取决于UserID有多大。

其次，要使用该方法实现良好的吞吐量，需要给UserID-to-PersonID索引提供特殊的硬件。为了实现合理的错误率，UserID集内仍至少需要100个元素。这意味着在查询中，你至少需要在UserID-to-PersonID索引中做100次查找。尽管相对于非取样方法需要的可能数以百万次的查找，这是一项巨大的改善，但这仍然是遭人诟病的。如果使用硬盘存储UserID-to-PersonID索引，那么每次到索引的查找至少需要一次磁盘寻道。由此可知，磁

盘寻道是多么地昂贵，且每个查询必须做这么多次磁盘寻道，这无疑将大大降低查询的吞吐量。

有两种方法可以打破这种瓶颈：一种方法是确保 UserID-to-PersonID 索引全部保存在内存中，这完全避免了去磁盘查找的需要。由于这种方法取决于索引的大小，因此它可能可行，也可能不可行；另一种方法是为了避免寻道和增加吞吐量，你会想要使用固态硬盘。需要特殊硬件来实现合理的吞吐量是这种方法的一个主要警告。

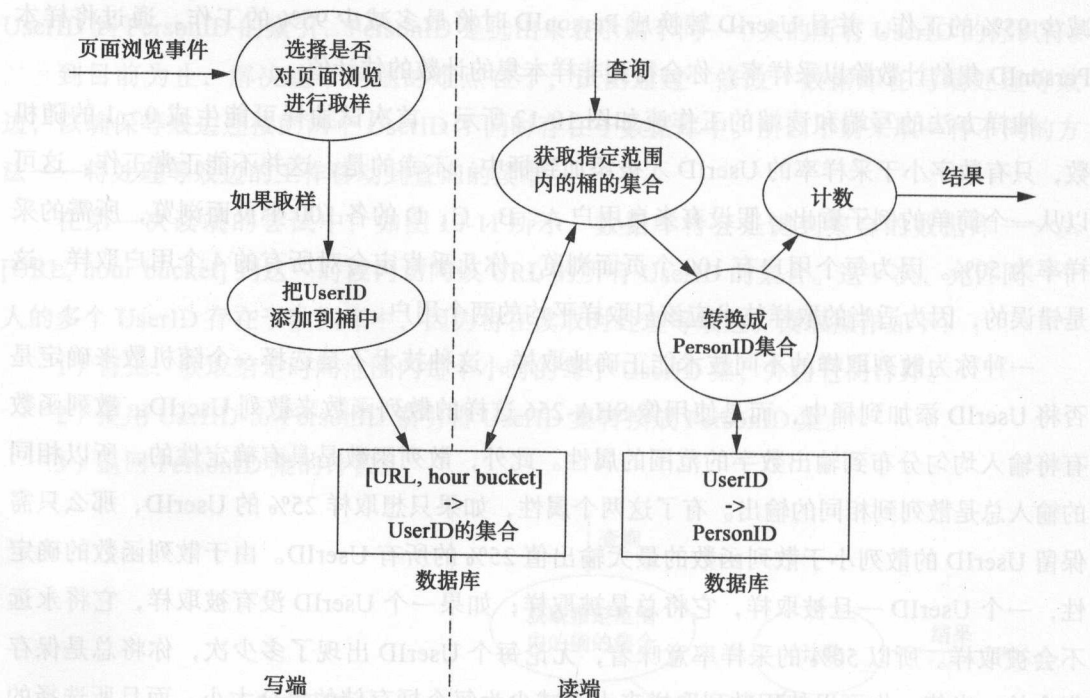


图 10-12 在工作流的读端添加取样

10.5.2 与 Lambda 架构解决方案的比较

带有等效边的给定时间范围内的独立访客的全增量解决方案，在各个方面都逊于 Lambda 架构解决方案。它必须使用错误率显然更高的近似技术、有更糟糕的延迟且需要特殊的硬件来实现合理的吞吐量。值得一问的是，为什么 Lambda 架构解决方案能够更加高效和简单？

造成所有这些差异的原因在于：一方面，批处理层有一次性查看所有数据的能力。全增量的解决方案必须在等效边传入时就处理它们，这妨碍了 HyperLogLog 的使用。另一方面，在批处理层，等效边被首先处理——通过将 UserID 规范化到 PersonID——然后采用

这种方式就创建了给定时间范围内的独立访客的视图。通过在前端处理等效边，你能够为给定时间范围内的独立访客的视图使用更有效的策略。之后，等效边必须在速度层被处理，但你会看到批处理层的存在使得这个问题更容易得多。

10.6 总结

本章介绍了 Lambda 架构服务层的基本概念：

- ❑ 定制视图来优化延迟和吞吐量的能力
- ❑ 不支持随机写的简单形式
- ❑ 在批处理层存储规范化数据并在服务层存储非规范化数据的能力
- ❑ 服务层固有的容错性和纠错性，因为它可以从主数据集被重新计算

完全根据服务层视图服务的查询来定制服务层视图的灵活性，是简单性的一个很好的示例。在传统的数据架构中，单个数据库被用作主数据集、历史存储和实时存储。为了满足这一要求，应用程序开发人员需要做出不受欢迎的取舍，比如怎样规范化和非规范化模式，还要承担主要的操作负担（比如处理压缩）。然而在 Lambda 架构中，这些角色都是由单独的组件所处理的。因此，每个角色可以进一步优化，并且系统作为一个整体更具鲁棒性。

第 11 章将给出实际的服务层数据库的示例。

服务层：示例

本章内容

- 作为服务层数据库示例的 ElephantDB
- ElephantDB 的体系结构
- ElephantDB 的缺点
- 为 SuperWebAnalytics.com 使用 ElephantDB

介绍了服务层的需求之后，本章将介绍如何构建一个专门作为服务层数据库使用的数据库的示例。与所有示例章节一样，本章没有介绍新的理论，而是你学过的概念到实际工具的细节的映射。

前面提到，服务层可用的工具落后于理想的可能性，当构建 SuperWebAnalytics.com 的服务层时，这一事实将非常明显。本章将介绍一个键/值服务层数据库——ElephantDB。因为它不支持键/值之外的类型索引，所以你必须从第 10 章所描述的理想索引类型中脱离出来。

我们将学习 ElephantDB 的基本架构，来理解它是如何满足服务层需求的，然后探索它的 API 来检索一个批处理视图的内容。最后，你将看到如何使用 ElephantDB 来索引和服务 SuperWebAnalytics.com 的批处理视图。

11.1 ElephantDB 的基本概念

ElephantDB 是一个键/值数据库——键和值都作为字节数组被存储。ElephantDB 将

批处理视图分区到固定数量的分片上，并且每台 ElephantDB 服务器负责这些分片的一些子集。

分配键到分片的函数是可插拔的，被称为**分片模式**。一个通用的模式通过获取键的散列值并用该值除以分片总数量的余数（模操作），来确定目标分片。我们将这种技术通俗地称为**散列取模**。它将键均匀地分配在分片之间，并提供了一种简单的方法来确定哪个分片托管一个给定的键。这通常是最好的选择，但你会遇到想要自定义切分方案的情况。一旦键/值被分配给一个分片，它就被存储在一个本地索引引擎上。在默认情况下，引擎是 BerkeleyDB，但引擎是可配置的，并且任意键/值索引引擎都可以在单台机器上运行。

ElephantDB 有两个功能：视图创建和视图服务。视图创建发生在批处理层 workflow 结束时的 MapReduce 作业中，所生成的分区存储在分布式文件系统中。然后由专用的 ElephantDB 集群来服务视图，该集群从分布式文件系统中加载分片，并与支持随机读取请求的客户端进行交互。在最终深入使用 ElephantDB 之前，我们将简要讨论这两个功能。

11.1.1 ElephantDB 中的视图创建

ElephantDB 分片是由输入为一组键/值对的 MapReduce 作业创建的。reducer 的数量被配置为 ElephantDB 分片的数量，并且使用指定的切分方案将键划分到 reducer。因此，每个 reducer 只负责生成一个分片的 ElephantDB 视图。然后，每个分片被索引（例如 BerkeleyDB 索引）并上传到分布式文件系统中。

注意：视图创建过程并不直接把分片发送到 ElephantDB 服务器。因为面向客户端的机器无法管理自己的负载，也无法应对可能损失的查询性能，所以这样的设计将是很差的。相反，ElephantDB 服务器以最快的速度从文件系统中抽取分片，使得它们保持向客户端承诺的性能。

11.1.2 ElephantDB 中的视图服务

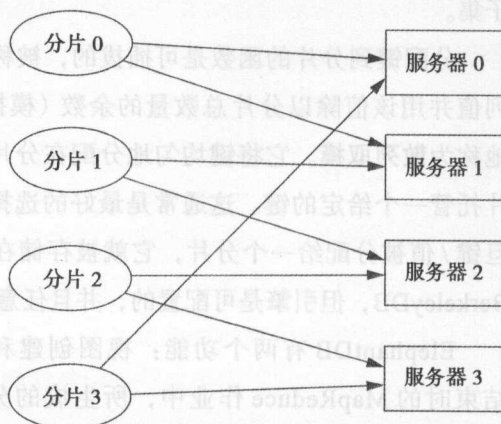
ElephantDB 集群由若干机器组成，每台机器分担了服务分片的工作。为了公平地分担负载，分片是被均匀分布在服务器之间的。

ElephantDB 还支持复制机制，每个分片在预定数量的服务器上被冗余地托管。例如，有 40 个分片，8 台服务器，复制因子为 3，每个服务器将托管 15 个分片，并且每个分片都存在于 3 台不同的服务器上。这使得集群容忍机器故障，即使机器出现故障，也允许完全访问整个视图。当然，只有在足够数据的机器同时出现故障，才会导致部分视图不可用，

但复制机制使得这种可能性非常小。复制机制如图 11-1 所示。

ElephantDB 服务器负责从分布式文件系统中检索它们指定的分片。当服务器检测到分片的新版本可用时，它就完成新分区的最快下载。下载是受控制的，这是为了不使机器的 I/O 过饱和且不影响实时读取。完成下载后，服务器就切换到新的分区并删除旧的分区。

ElephantDB 服务器下载完分片之后，批处理视图的内容是通过一个基本的 API 来访问的。前面提到过没有通用的服务层数据库——这使得 ElephantDB 的局限性越来越明显。因为 ElephantDB 使用键/索引的模型，API 只允许检索指定键的值。通用的服务层数据库将提供更丰富的 API，如扫描键的范围。



利用复制策略，将每个分片都存在于多台服务器上。

图 11-1 复制机制使得分片存储在多个位置，以容忍个别机器故障

11.1.3 使用 ElephantDB

ElephantDB 的简单性使它极易使用。使用 ElephantDB 涉及三个不同的方面：创建分片、建立服务请求的集群以及使用客户端 API 来查询批处理视图。

1. 创建 ElephantDB 分片

使用 JCascalog 的 tap 抽象使得创建一组 ElephantDB 分片非常简单。ElephantDB 提供了一个 tap 来自动创建分片。如果你有一个生成键/值对的子查询，那么创建 ElephantDB 视图就像执行子查询到 tap 中一样简单：

使用 Berkeley-DB 作为本地存储引擎。

```
public static elephantDbTapExample (Subquery subquery) {
    DomainSpec spec = new DomainSpec(new JavaBerkDB(),
                                     new HashModScheme());
    Object tap = EDB.makeKeyValTap("/output/path/on/dfs", spec, 32);
    Api.execute(tap, subquery);
}
```

应用散列取模分区作为分片方案。

将子查询的输出指引到构造的 tap 中。

对给定的分布式文件系统路径创建 32 个分片。

在底层，已配置的 tap 通过自动配置 MapReduce 作业来正确地键进行分区，创建每个索引，并将每个索引上传到分布式文件系统中。

2. 建立 ElephantDB 集群

建立 ElephantDB 集群有两个需要的配置: 一个本地配置和一个全局配置。本地配置包含服务器的特定属性, 以及全局配置和实际分片所在的地址。一个基本的本地配置位于每台服务器上, 示例如下:

```
{:local-root "/data/elephantdb"
:hdbs-conf {"fs.default.name" "hdfs://namenode.domain.com:8020"}
:blob-conf {"fs.default.name" "hdfs://namenode.domain.com:8020"}}
```

存储下载的分片的本地路径。

存储分片的分布式文件系统的地址。

托管全局配置的分布式文件系统的地址。

全局配置包含集群中每一台服务器所需要的信息, 其中包括复制因子、服务器应该用来接受请求的 TCP 端口以及集群服务的视图。单个集群可以服务多个域, 所以该配置包含从域名到它们的 HDFS 位置的映射。

一个基本的全局配置的代码如下:

```
{:replication 1
:hosts ["edb1.domain.com" "edb2.domain.com" "edb3.domain.com"]
:port 3578
:domains {"tweet-counts" "/data/output/tweet-counts-edb"
"influenced-by" "/data/output/influenced-by-edb"
"influencer-of" "/data/output/influencer-of-edb"}}
```

对于所有服务器的所有视图的复制因子。

托管集群中所有服务器的名字。

服务器将用来接受请求的 TCP 端口。

分布式文件系统中所有视图的标识符和位置。

这些配置是如此简单, 以至于它们几乎显得不那么完整。例如, 从服务器到配置将要托管的指定分片是没有明确分配的。在这个具体的示例中, 服务器以托管列表中它们的位置, 作为一个确定性函数的输入, 来计算它们应该下载的分片。配置的简单性反映了使用 ElephantDB 的轻松。

怎样真正启动 ElephantDB 服务器?

启动 ElephantDB 服务器的过程遵循标准的 Java 实践, 比如建立一个项目 jar 包, 并通过命令行语句传递配置的位置。本书不会提供可能很快过时的细节, 而是建议你参考项目网站 (<http://github.com/nathanmarz/elephantdb>) 来了解更多细节。

3. 查询 ElephantDB 集群

ElephantDB 公开了一个发起查询的简单 Thrift API。连接到任意 ElephantDB 服务器之

后，你可以发起如下的查询：

```
public static void clientQuery(ElephantDB.Client client,
                               String domain,
                               byte[] key) {
    client.get(domain, key);
}
```

如果所连接的服务器本地没有存储请求的键，它将通过与集群中的其他服务器进行通信来检索所需的值。

11.2 创建 SuperWebAnalytics.com 的服务层

介绍了基础知识之后，现在你可以为 SuperWebAnalytics.com 中的每个查询创建优化的 ElephantDB 视图。首先是给定时间范围内的页面浏览量视图。

11.2.1 给定时间范围内的页面浏览量

回想一下，给定时间范围内的页面浏览量的理想视图是从键排序映射的索引，如图 11-2 所示。可以发现，最大的时间粒度是以小时为单位的，因为每一项只需要几个字节的存储，所以扫描若干年时间范围的数据是相当容易的。

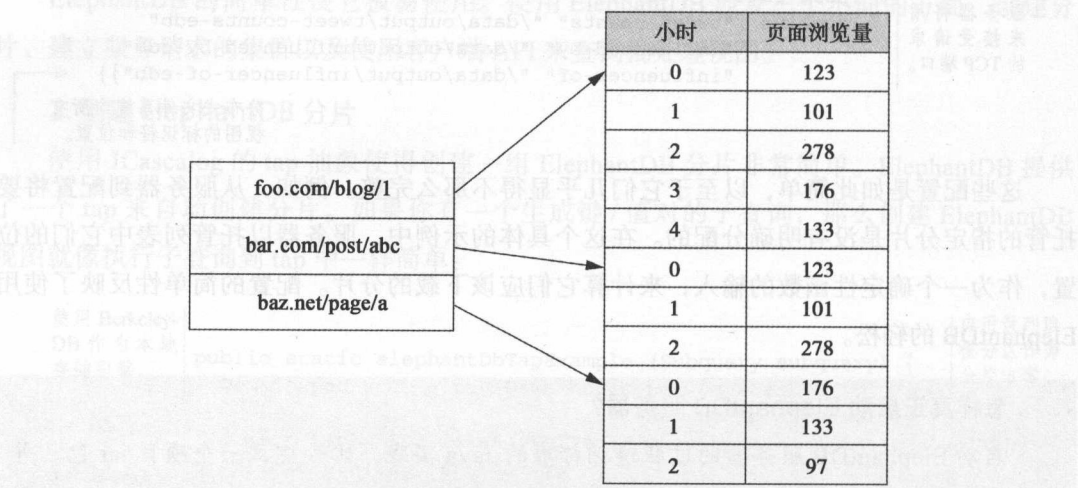


图 11-2 给定时间范围内的页面浏览量的理想索引策略

不幸的是，ElephantDB 只支持键 / 值索引，所以这种视图对于 ElephantDB 是不可能的。因为每个键需要单独被检索，所以为每个查询减少检索的键的数量是很有必要的。这意味着所有粒度都应该被索引到视图中。下面来看如何使用 ElephantDB 来实现该

策略。

第 8 章结束时，你已经生成了如图 11-3 所示的视图。回想一下，ElephantDB 中的键和值被作为字节数组存储。对于给定时间范围内的页面浏览量视图，你需要将 URL、粒度和时间桶编码到键中。

URL	粒度	桶	页面浏览量
foo.com/blog/1	h	0	10
foo.com/blog/1	h	1	21
foo.com/blog/1	h	2	7
foo.com/blog/1	w	0	38
foo.com/blog/1	m	0	38
bar.com/post/a	h	0	213
bar.com/post/a	h	1	178
bar.com/post/a	h	2	568

图 11-3 给定时间范围内的页面浏览量的批处理视图

下面的 JCascalog 函数实现了组合键和页面浏览值所需要的序列化：

```
public static class ToUrlBucketedKey extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        String url = call.getArguments().getString(0);
        String gran = call.getArguments().getString(1);
        Integer bucket = call.getArguments().getInteger(2);

        String keyStr = url + "/" + gran + "-" + bucket;
        try {
            call.getOutputCollector()
                .add(new Tuple(keyStr.getBytes("UTF-8")));
        } catch (UnsupportedEncodingException e) {
            throw new RuntimeException(e);
        }
    }
}

public static class ToSerializedLong extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        long val = call.getArguments().getLong(0);
        ByteBuffer buffer = ByteBuffer.allocate(8);
        buffer.putLong(val);
        call.getOutputCollector().add(new Tuple(buffer.array()));
    }
}
```

将键的组成部分
连接在一起。

使用 UTF-8 编码
转换成字节。

配置 Byte-Buffer
来保存单个长整
型值。

从缓冲区中提取
字节数组。

下一个步骤是创建 ElephantDBtap。为了归避本章开始讨论的方差问题，你可以创建一

个自定义的 `ShardingScheme`，来确保单个 URL 的所有键 / 值对都存在于相同的分片中。以下代码片段通过对组合键的 URL 部分进行散列取模实现了该任务：

```
private static String getUrlFromSerializedKey(byte[] ser) {
    try {
        String key = new String(ser, "UTF-8");
        return key.substring(0, key.lastIndexOf("/"));
    } catch (UnsupportedEncodingException e) {
        throw new RuntimeException(e);
    }
}

public static class UrlOnlyScheme implements ShardingScheme {
    public int shardIndex(byte[] shardKey, int shardCount) {
        String url = getUrlFromSerializedKey(shardKey);
        return url.hashCode() % shardCount;
    }
}
```

从组合键中提取 URL。

返回 URL 的散列模数。

下面的 `JCasalog` 子查询将所有这些部分组合在一起，使批处理层视图转换成适合于 `ElephantDB` 的键 / 值对：

```
public static void pageviewElephantDB(Subquery batchView) {
    Subquery toEdb =
        new Subquery("?key", "?value")
            .predicate(batchView, "?url", "?gran", "?bucket", "?total-views")
            .predicate(new ToUrlBucketedKey(), "?url", "?gran", "?bucket")
            .out("?key")
            .predicate(new ToSerializedLong(), "?total-views")
            .out("?value");

    DomainSpec spec = new DomainSpec(new JavaBerkDB(),
                                      new UrlOnlyScheme(),
                                      32);

    Tap tap = EDB.makeKeyValTap("/outputs/edb/pageviews", spec);
    Api.execute(tap, toEdb);
}
```

子查询必须只返回与键和值对应的两个字段。

定义本地存储引擎、分片方案和分片的总数量。

指定分片的 HDFS 位置。

执行转换。

此外，给定时间范围内的页面浏览量视图将受益于一个更通用的服务层数据库，该数据库可以为每个 URL 按时间顺序存储时间桶。该数据库将利用磁盘扫描，并将昂贵的磁盘寻道减至最少。

虽然创建一个这样的数据库比大多数现有的 NoSQL 数据库简单得多，但在撰写本书时，这样一个服务层数据库是不存在的。不过这里展示的方法并不比理想的服务层数据库差，因为它仍能确保单个查询的所有索引检索只与一个节点通信，并且对于任何给定的查询，它只需要获取少量的值。

11.2.2 给定时间范围内的独立访客数量

下一个查询是给定时间范围内的独立访客数量查询。与给定时间范围内的页面浏览量一样, 服务层数据库中缺乏键排序映射 (Key-to-sorted-map) 阻碍了前面章节中描述的理想索引的实现。但是可以使用与给定时间范围内的页面浏览量的类似策略, 来生成一个可行的解决方案。

两个查询之间的唯一区别是, 给定时间范围内的独立访客数量存储 HyperLogLog 集。与给定时间范围内的页面浏览量一样, 为了避免方差问题, 给定时间范围内的独立访客数量可以利用相同的分片方案。下面是生成给定时间范围内的独立访客数量视图的代码:

```
public static void uniquesElephantDB(Subquery uniquesView) {
    Subquery toEdb =
        new Subquery("?key", "?value")
            .predicate(uniquesView, "?url", "?gran", "?bucket", "?value")
            .predicate(new ToUrlBucketedKey(), "?url", "?gran", "?bucket")
            .out("?key");
    DomainSpec spec = new DomainSpec(new JavaBerkDB(),
                                      new UrlOnlyScheme(),
                                      32);
    Tap tap = EDB.makeKeyValTap("/outputs/edb/uniques", spec);
    Api.execute(tap, toEdb);
}
```

只有组合键需要被序列化, 因为 HyperLogLog 集已经被序列化了。

为独立访客页面浏览数量的分片改变输出目录

理想的服务层数据库将会知道如何在本机上处理 HyperLogLog 集, 并在服务器上完成查询。服务器将 HyperLogLog 集合合并, 并返回 HyperLogLog 结构的基数, 而不是直接返回查询数据库得到的 HyperLogLog 集。通过避免查询期间的任何 HyperLogLog 集的网络传输, 可以最大限度地提高效率。

11.2.3 跳出率分析

理想的跳出率分析视图是键/值索引, 所以可以用 ElephantDB 生成理想的视图。跳出率分析视图是从每个域到访问次数和反弹次数的映射。

你可以重用以前的查询的框架, 但仍然需要为字符串的键和组合的值自定义序列化代码:

```
public static class ToSerializedString extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        String str = call.getArguments().getString(0);
        try {
            call.getOutputCollector().add(new Tuple(str.getBytes("UTF-8")));
        } catch (UnsupportedEncodingException e) {
            throw new RuntimeException(e);
        }
    }
}
```

该序列化函数本质上与组合键的序列化函数是相同的。

```

public static class ToSerializedLongPair extends CaskalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        long l1 = call.getArguments().getLong(0);
        long l2 = call.getArguments().getLong(1);
        ByteBuffer buffer = ByteBuffer.allocate(16);
        buffer.putLong(l1);
        buffer.putLong(l2);
        call.getOutputCollector().add(new Tuple(buffer.array()));
    }
}

```

为两个长整型值分配空间。

查询该视图时一次只能获取一个域，因此在服务器响应时间方面不用担心方差问题。因此，正常的散列取模分片是适合这种情况的。代码如下：

```

public static void bounceRateElephantDB(Subquery bounceView) {
    Subquery toEdb =
        new Subquery("?key", "?value")
            .predicate(bounceView, "?domain", "?bounces", "?total")
            .predicate(new ToSerializedString(), "?domain")
            .out("?key")
            .predicate(new ToSerializedLongPair(), "?bounces", "?total")
            .out("?value");

    DomainSpec spec = new DomainSpec(new JavaBerkDB(),
                                      new HashModScheme(),
                                      32);

    Tap tap = EDB.makeKeyValTap("/outputs/edb/bounces", spec);
    Api.execute(tap, toEdb);
}

```

使用 ElephantDB 提供的散列取模分片方案。

如你所见，将批处理视图集成到服务层几乎不用做什么工作。

11.3 总结

ElephantDB 是一个可以用于服务层的数据库。可以看到，ElephantDB 的使用和操作非常简单。我们希望看到根据不同的或更通用的索引模型来创建的其他服务层数据库，因为服务层的基础简单性使得这些数据库很容易构建。

现在你已经了解了批处理层和服务层，接下来是学习 Lambda 架构的最后一部分——速度层。速度层将弥补服务层的高延迟更新，并允许查询访问最新的数据。

弃。尽管速度层更为复杂且更容易出错，但任何错误都是预期的，并将通过更简单的批处理层和服务层自动纠正。

正如之前一再声明的，Lambda 架构的厉害之处在于不同角色的分离（见图 12-4），

基于关系型数据库（中，所有这些结构都存在速度层。这种特性方面的选择十分有限。

章 12 第

第三部分 *Part 3*

速度层

第三部分主要介绍 Lambda 架构的速度层。速度层弥补了批处理层的高延迟，使得查询能获取最新的结果。

第 12 章讨论实时视图和批处理视图。二者的主要区别是实时视图数据库必须支持随机写，这大大增加了数据库的复杂性。你会发现批处理层的存在减轻了管理这样一个数据库的复杂性。你还将看到速度层可以同步或异步地实现。

第 13 章为示例章节，展示了使用 Apache Cassandra 的实时视图。

同步架构不需要过多解释，所以第 14 章开始讨论速度层的异步架构，主要讨论使用队列和流处理的增量计算。流处理主要有两种模式：一次一个流处理和微批量流处理，这两种模式各有利弊。第 14 章探讨了一次一个的流处理的概念；第 15 章为示例章节，展示了使用 Apache Kafka 和 Apache Storm 的模型。

第 16 章深入探讨了另一种模式——微批量流处理。你会发现，通过牺牲一些延迟，你将获得强大的新功能。第 17 章为示例章节，展示了使用 Trident 的微批量流处理。

实时视图

本章内容

- ❑ 速度层的理论模型
- ❑ 批处理层如何简化速度层的职责
- ❑ 对实时视图使用随机写数据库
- ❑ CAP 定理及其意义
- ❑ 增量计算的挑战
- ❑ 速度层数据的过期处理

到目前为止，我们围绕着批处理层和服务层对 Lambda 架构展开了讨论——批处理层和服务层是涉及在每一片数据上计算函数的组件。这些层满足了数据系统除了低延迟更新属性以外的所有所需的属性。而速度层的唯一工作就是满足这最后的需求。

在整个主数据集上运行函数——可能是 PB 级别的数据——是资源密集型的操作。为了以尽可能低的延迟更新，速度层必须从根本上采取一种与批处理层和服务层不同的方法。因此，速度层是基于增量计算而不是批量计算的。

增量计算引入了许多新的挑战，而且比批量计算更复杂。幸运的是，速度层有限的要求提供了两个优势：首先，速度层只负责没有包含在服务层视图中的数据——数据最多是几个小时之内产生的，且远远小于主数据集，而处理规模较小的数据允许更大的设计灵活性。其次，速度层视图是暂时的。数据一旦被获取到服务层视图，它就可以从速度层被丢

弃。尽管速度层更为复杂且更容易出错，但任何错误都是短期的，并将通过更简单的批处理层和服务层自动纠正。

正如之前一再声明的，Lambda 架构的厉害之处在于不同层角色的分离（见图 12-1）。在传统的数据架构（比如那些基于关系型的数据库）中，所有这些结构都存在速度层。这些系统在处理增量计算的复杂性方面的选择十分有限。

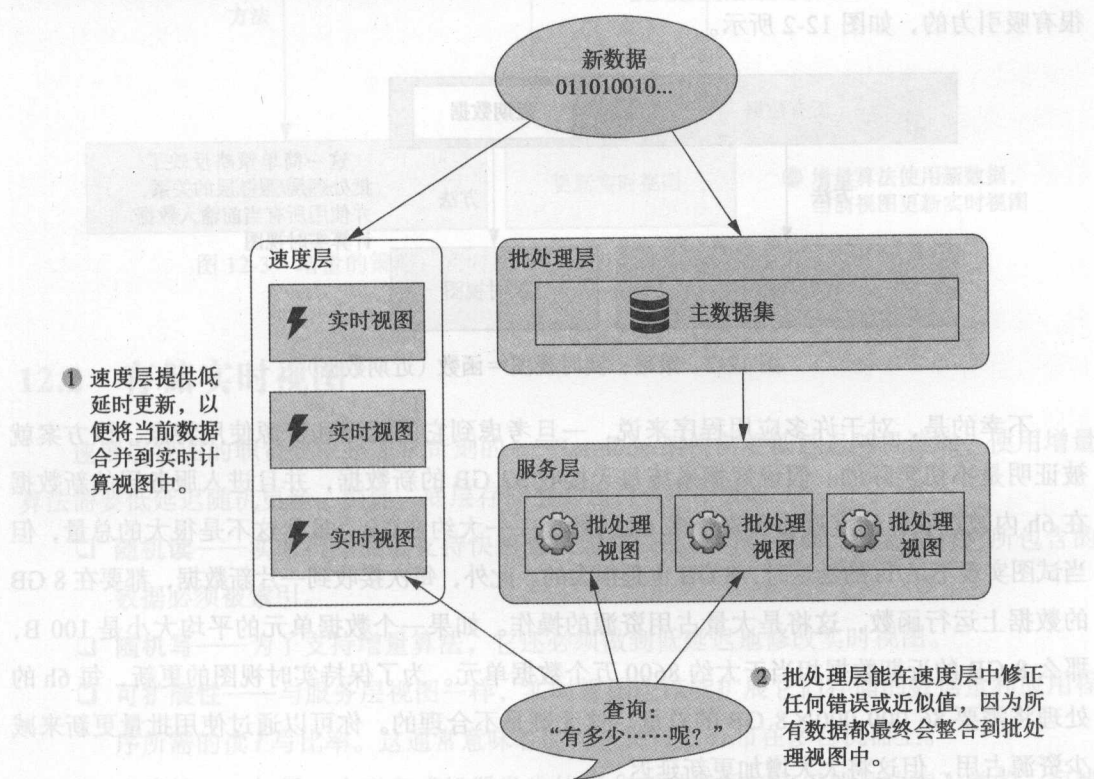


图 12-1 速度层允许 Lambda 架构在最新的数据上以低延迟服务查询

速度层有两个主要功能：存储实时视图；处理传入的数据流以便更新这些视图。本章主要介绍实时视图的结构和存储：首先概述速度层的理论基础，其次继续讨论你会遇到的增量计算的各种挑战，最后演示如何对速度层的数据进行过期处理。

12.1 计算实时视图

速度层的基本目标与批处理层和服务层是一样的，即生成可以被高效查询的视图。关键的不同之处在于，速度层中的这些视图只代表近期的数据，并且它们必须在新数据到达

后很快被更新。“很快”的含义对每个应用程序都不同，但它通常是指几毫秒到几秒。这个要求对生成速度层视图的计算方法有着深远的影响。

为了理解这种影响，你不妨考虑使用这样一种简单的方法：类似于批处理层和服务层通过在整个主数据集上计算方法来生成视图，速度层可以通过在所有近期数据（即还未进入服务层的数据）上运行方法来生成它的视图。速度层的简单性和与批处理层工作的一致性是很有吸引力的，如图 12-2 所示。

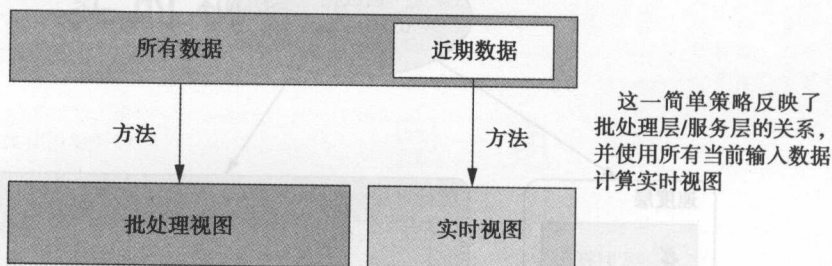


图 12-2 策略：实时视图=函数(近期数据)

不幸的是，对于许多应用程序来说，一旦考虑到它的延迟和资源使用特点，该方案就被证明是不切实际的。假设数据系统每天接收 32 GB 的新数据，并且进入服务层的新数据在 6h 内被接收。速度层将负责最多 6h 的数据——大约 8 GB。虽然这不是很大的总量，但当试图实现不足 1s 的延迟时，8 GB 也是很大的。此外，每次接收到一片新数据，都要在 8 GB 的数据上运行函数，这将是大量占用资源的操作。如果一个数据单元的平均大小是 100 B，那么 8 GB 的近期数据相当于大约 8600 万个数据单元。为了保持实时视图的更新，每 6h 的处理将需要 $86\,000\,000 \times 8\text{ GB}$ 的总量，这无疑是不合理的。你可以通过使用批量更新来减少资源占用，但这将大大增加更新延迟。

如果应用程序可以接受几分钟的延迟，那么这种简单的策略就是一种很好的方法。但通常来说，你需要以高资源利用率的方式生成毫秒级别延迟的实时视图。在本章的其余部分，我们将针对这种情况展开讨论。

一般来说，任何可行的解决方案都依赖于所使用的增量算法，如图 12-3 所示。该想法是为了在数据传入时更新实时视图，从而重用之前用于生成视图的操作。这需要使用随机读/随机写的数据库，这样就可以在现有的视图上执行更新操作。在 12.2 节中，我们将进一步讨论这些数据库，同时深入研究速度层视图的存储细节。

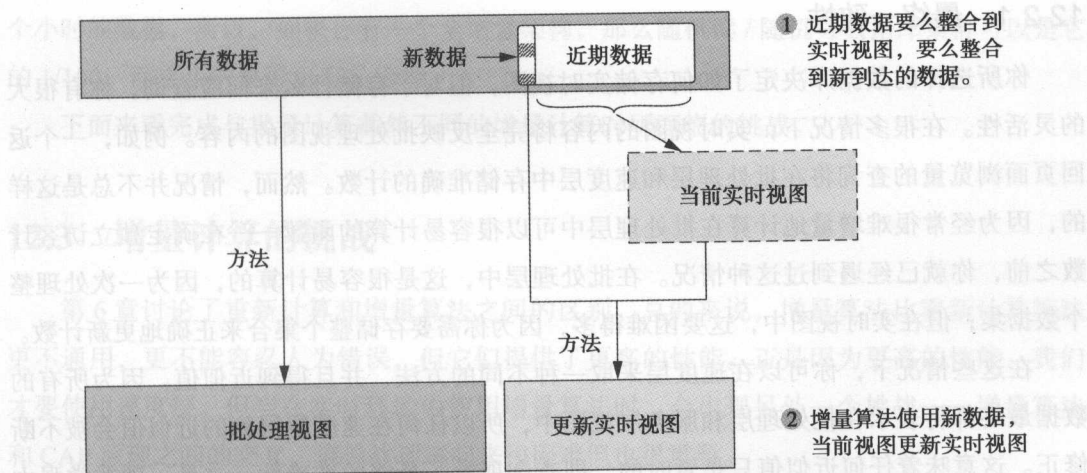


图 12-3 增量的策略：实时视图 = 方法（新数据，之前的实时视图）

12.2 存储实时视图

速度层视图的职责要求是非常苛刻的——Lambda 架构需要低延迟随机读取，使用增量算法需要低延迟随机更新。因此，底层存储层必须符合以下要求：

- ❑ 随机读——实时视图应该支持快速随机读取以迅速回应查询。这意味着它所包含的数据必须被索引。
- ❑ 随机写——为了支持增量算法，它还必须做到低延迟地修改实时视图。
- ❑ 可扩展性——与服务层视图一样，实时视图应该能扩展它们存储的数据量和应用程序所需的读 / 写比率。这通常意味着实时视图可以分布在多台机器上。
- ❑ 容错性——如果一个磁盘或机器发生故障，实时视图应该仍然是功能正常的。容错性是通过在机器之间复制数据来实现的，因此，即使单台机器发生故障，实时视图也是有备份的。

这些属性对于被称为 NoSQL 数据库的一类数据库是很常见的。NoSQL 数据库支持不同的数据模型和索引类型，所以你可以选择一个或多个实时视图数据库来满足自己的索引需求。例如，你可以选择 Cassandra 来存储键 / 值格式的索引，然后使用 ElasticSearch 存储支持搜索查询的索引。最终，你可以非常灵活地选择数据库的组合，以满足速度层的精确需求。

12.2.1 最终一致性

你所选择的数据库决定了如何存储实时视图，但对于存储什么来回应查询，你有很大的灵活性。在很多情况下，实时视图的内容将完全反映批处理视图的内容。例如，一个返回页面浏览量的查询将在批处理层和速度层中存储准确的计数。然而，情况并不总是这样的，因为经常很难增量地计算在批处理层中可以很容易计算的函数——在确定独立访客计数之前，你就已经遇到过这种情况。在批处理层中，这是很容易计算的，因为一次处理整个数据集；但在实时视图中，这要困难得多，因为你需要存储整个集合来正确地更新计数。

在这些情况下，你可以在速度层采取一种不同的方法，并且得到近似值。因为所有的数据最终都将表示在批处理层和服务层视图中，所以任何在速度层得到的近似值会被不断修正。这意味着任何近似值只是暂时的，即查询展示了最终的准确性。这是一项非常强大的技术，提供了所有优点：性能、准确性和及时性。最终准确的方法对于复杂查询是常见的，如那些需要实时机器学习的查询。这种视图倾向于将多种数据关联在一起，无论以何种合理的方式，它都不能增量地生成一个准确值。当具体化 SuperWebAnalytics.com 的速度层时，你将看到利用最终准确性的示例。

必须再次强调的是，最终准确性是一项可选的技术，可以用于显著降低速度层的资源占用。因为 Lambda 架构的两个层有不同的计算方法，所以这仅仅是一个可用的选项。在基于全增量计算的传统架构中，这种技术就不是可选的了。

12.2.2 速度层中存储的状态总量

速度层存储相对少量的状态，因为它们只表示近期数据的视图。这是一项优势，因为实时视图比服务层视图更复杂。

下面简要回顾一下服务层所绕过的实时视图的复杂性：

- ❑ **在线压缩**——当读/写数据库接收到更新时，部分磁盘索引会成为闲置的、浪费的空间。数据库必须定期地执行压缩以回收空间。压缩是一个资源密集型过程，可能会“饿死”需要快速服务查询的资源的机器。不恰当的压缩管理会导致整个集群的级联失败。
- ❑ **并发性**——读/写数据库可能会同时收到同一个值的多次读或写操作。因此需要协调这些读写操作，以防返回过期或不一致的值。跨线程共享可变状态是一个众所周知的复杂性问题，并且像锁这样的控制策略，其易出错的特点也是众所周知的。

注意到速度层的压力没那么大是很重要的，因为比起服务层它存储了远远少得多的数据。Lambda 架构中角色和职责的分离限制了速度层中的复杂性。因为速度层通常只负责几

个小时的数据，所以，如果你有一个全增量架构，那么随机读 / 随机写数据库集群可以是它的 1/100，而更小的集群更容易管理。

下面来看完成与批量计算截然不同的增量计算时所面临的挑战。

12.3 增量计算的挑战

第 6 章讨论了重新计算和增量算法之间的区别。总的来说，增量算法比重重新计算算法更不通用，更不能容忍人为错误，但它们提供了更高的性能。正是因为更高的性能，我们才要使用速度层。但当在实时环境中使用增量算法时，会出现另外一个挑战——增量算法和 CAP 原理之间的交互。该挑战理解起来很困难但也很重要。

CAP 原理与一致性和可用性之间的基本权衡有关，一致性中的读操作被保证合并所有之前的写操作，可用性中的每一个查询返回的是正确答案而不是错误消息。不幸的是，该定理经常被以令人误解和不准确的方式进行解释。我们通常会避免呈现任何不准确的解释，但这种解释是如此普遍，所以有必要就该解释讨论一下，以澄清误解。

CAP 通常被解释为“你最多可以有一致性、可用性和分区容忍性中的两个特性。”这种解释的问题在于，当不是所有机器都能相互通信时，CAP 原理完全是“关于数据系统发生了什么”的原理。认为数据系统是一致和可用的但不是分区容忍是毫无意义的，因为该原理完全是“关于分区情况下发生了什么”的原理。

正确表述 CAP 原理的方式是“当分布式数据系统被分区时，它可以是一致性的或可用性的，但不能两者兼而有之。”也就是说，如果选择一致性，那么查询有时会收到错误消息而不是正确答案；如果选择可用性，那么在网络分区期间，读操作可能返回过时的结果。在高可用性系统中，你可以拥有的最好的一致性属性是最终一致性，在这种情况下，一旦网络分区结束，系统将返回到一致性。

12.3.1 CAP 原理的有效性

很容易理解为什么 CAP 原理是正确的。假设有一个简单的分布式键 / 值数据库，集群中的每个节点负责单独的一组键，这意味着没有复制。若想对特定的键进行读或写，则会有单台机器负责处理该请求。

现在假设你突然不能与分布式系统中的一些机器通信。显然，你不能对不可访问的节点进行读或写数据。

你可以通过跨多台服务器复制数据来增加分布式数据系统的容错性。对于复制因子为

3 的情况，每个键将被复制到 3 个节点。现在如果你从一个副本被分区出来，那么仍然可从另一个位置检索值。但它仍然可能从所有副本被分区出来，尽管这是不太可能的。

真正的问题是如何处理分区情况下的写操作。有几个选项——比如，你可以拒绝执行更新操作，除非所有副本都可以一次性被更新。利用该策略，每个读请求被保证返回最新的值。这是一种一致性的策略，因为在做读取操作时，从彼此分区出来的客户端总是收到相同的值（或得到一个错误）。

或者，你可以选择更新任何可用的副本，并在分区结束时同步其他副本。当发生如图 12-4 所示的情况时，这可能会比较麻烦。在这种情况下，客户端被不同地分区，并与机器的不同子组进行通信。如果想要更新副本的一个子集，那么副本将会产生差异，比起简单地选择最近的更新，合并是更复杂的。

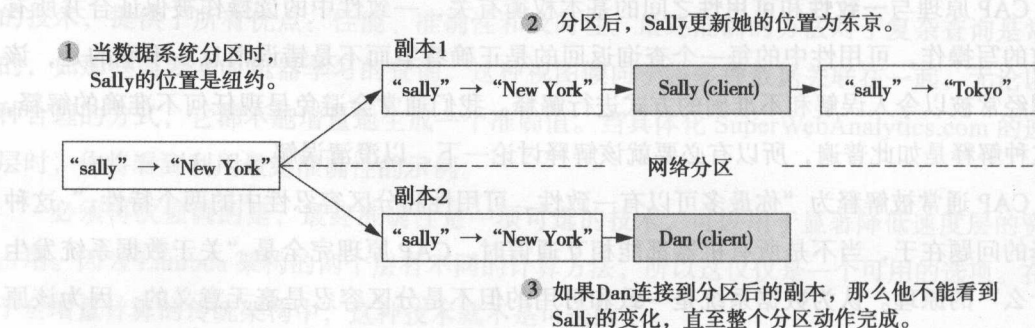


图 12-4 如果在分区的情况下允许更新，副本将会产生差异

利用局部更新策略，数据系统是可用的但不是一致的。在图 12-4 中，Sally 可能将她的位置更新为东京，但 Dan 的副本不能被更新，所以他将读取到一个过时的值。这个示例表明，在分区期间，不可能同时拥有可用性和一致性，因为无法与有最新信息的副本通信。

极高可用性——sloppy quorums

一些分布式数据库有一个称为 sloppy quorums 的选项，它提供了极高的可用性——即使副本对于该数据并不可用，也接受该数据的写操作。作为替代，一个临时副本将被创建，一旦正式副本可用，就将临时副本合并到正式副本。利用 sloppy quorums，如果每个节点是从其他节点被分区出来的，那么一块数据的可能副本数量与集群中节点的数量相等。虽然这可能是有用的，但请记住，这种做法增加了系统附带的复杂性。

批处理层和服务层是分布式系统，并且与其他系统一样受限干 CAP 原理。批处理层中

唯一的写操作是不可变数据的新增部分的写入。这些写操作不需要在机器之间进行协调，因为每片数据都是独立的。如果数据不能被写入批处理层的输入数据存储中，它可以在本地缓冲然后重试。至于服务层，由于批处理层的高延迟，读操作总是过时的。服务层不仅是不一致，由于它总是过时的，因此它甚至不是最终一致的。所以，批处理层和服务层都选择了可用性而不是一致性。

注意：在确定这些属性时不需要什么复杂的东西——批处理层和服务层的逻辑很简单，理解起来也很容易。不幸的是，当在实时环境中使用增量算法追求最终一致性时，事实并非如此。

12.3.2 CAP 原理和增量算法之间复杂的相互作用

正如刚刚所讨论的，如果你为分布式系统选择高可用性，那么一个分区将创建同一个值的多个副本，因为这个值在另外一个分区被单独更新。当分区结束时，这些值必须合并在一起，以便新值合并了分区期间的每次更新——不多也不少。问题是，对于每个用例，没有简单的方法来完成这项工作，所以识别工作策略的任务落在了开发人员身上。例如，考虑如何实现最终一致性的计算。

在此场景中，假定你只存储计数作为值。假设网络被分区，副本独立地变化，然后分区被修正。当合并副本值时，你发现一个副本计数为 110，另一个副本计数为 105。新的值应该是多少？困惑之处在于，你不确定它们什么时候开始出现分歧。如果它们在值为 105 时出现分歧，那么更新后的计数应该是 110。如果它们在值为 0 时出现分歧，那么正确的答案是 215。所有你确定知道的是，正确答案位于这两个界限之间。

为了正确地实现最终一致性的计算，你需要利用被称为 **conflict-free replicated data types** 的结构（通常称为 CRDTs）。各种值和操作的 CRDTs 有很多：只支持添加的集合、支持添加和删除的集合、支持增量的数值、支持增量和减量的数值等。G-Counter 是一类只支持增量的 CRDT，这正是解决当前计数问题所需要的。一个示例的 G-Counter 如图 12-5 所示。G-Counter 为每个副本存储一个不同的值，而不是一个单一的值。因此，实际的计数是副本计数的总和。

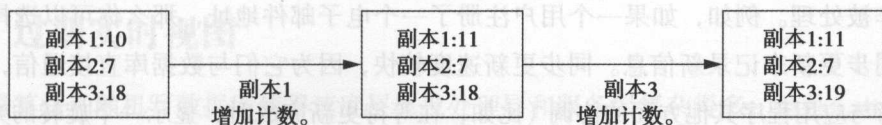


图 12-5 G-Counter 是只增加计数器，副本只增加分配给它的计数器。计数器的总体的值是副本计数的总和

如果副本之间检测到冲突，那么新的 G-Counter 将获取每个副本的最大值。因为计数是只增加的，而且系统中只有一台服务器将更新给定副本的计数，所以最大值确保是正确的值。如图 12-6 所示。

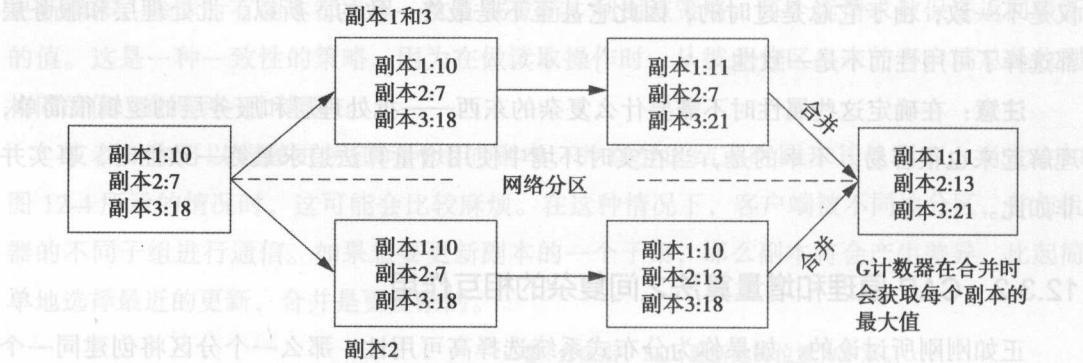


图 12-6 合并 G-Counter

如你所见，在实时的最终一致性环境下，实现计数更加复杂。保持一个简单的计数是不够的——你还需要一种策略，用以修复分区中有分歧的值。该算法也会引入更多的复杂性。如果允许减量和增量，那么数据结构和合并算法将变得更加复杂。这些合并算法通常被称为读修复算法，是人为错误的巨大来源。从复杂性的角度来讲，这并不奇怪。

不幸的是，如果想要实现速度层的最终一致性，那么这种复杂性是无法摆脱的。但是有一件事情是有益的——Lambda 架构的固有保护可以避免出错。如果由于忘记了一种边的情况或搞砸了合并算法而导致实时视图被损坏，那么之后批处理层和服务层将在服务层视图中自动纠正错误。就错误而言，最糟糕的可能结果是暂时的损坏。如果没有备份实时视图的批处理层的架构，那么增量部分将会是永久的损坏。

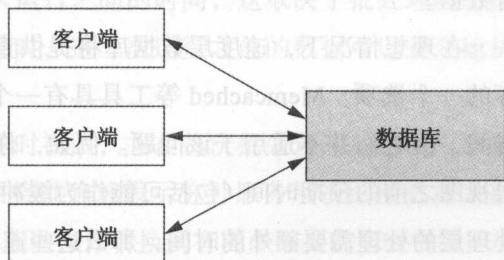
12.4 异步更新与同步更新

速度层体系结构的不同取决于实时视图是同步更新还是异步更新。

对于同步更新，你可能要做 100 万次：应用程序直接向数据库发出请求，并阻塞直到更新操作被处理。例如，如果一个用户注册了一个电子邮件地址，那么你可以选择向数据库发起同步更新来记录新信息。同步更新速度较快，因为它们与数据库直接通信，并且促进了更新与应用程序其他方面的协调（比如，在等待更新完成时，显示一个旋转的光标）。

用于同步更新的速度层体系结构如图 12-7 所示。不出意料，应用程序简单地直接向数据库发起了更新。

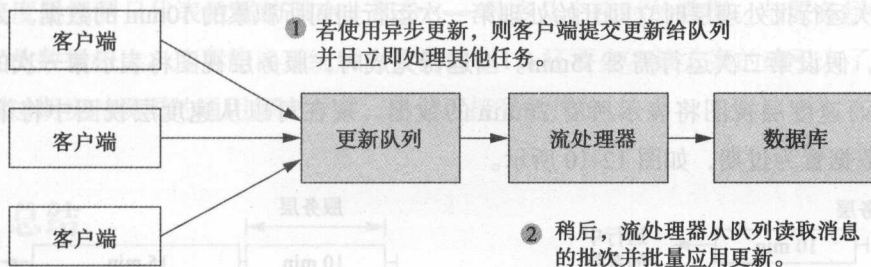
相比之下，异步更新请求被放置在一个队列中，并稍后发生更新。在速度层中，该延迟的可能范围从几毫秒到几秒钟，但如果有过多的请求，它可能需要更长的时间。异步更新慢于同步更新，因为在数据库被修改之前，它们需要额外的步骤，并且不可能与其他操作相协调，因为你不能控制它们什么时候执行。但异步更新提供了许多优势：首先，你可以从队列中读取多个消息并向数据库执行批量更新，这极大地提高了吞吐量。其次，它们对于不同负载的处理也很容易——如果更新请求的数量飙升，那么队列将缓冲额外的请求，直到所有更新被执行。相反，同步更新的请求高峰可能使数据库过载，导致请求丢失、超时和其他破坏应用程序的错误。



若使用同步更新，则客户端直接与数据库通信并阻塞直到更新完成

图 12-7 使用同步更新的简单的速度层体系结构

用于异步更新的速度层体系结构如图 12-8 所示。在第 13 章中，你将从更多细节上了解队列和流处理。



① 若使用异步更新，则客户端提交更新给队列并且立即处理其他任务。

② 稍后，流处理器从队列读取消息的批次并批量应用更新。

图 12-8 异步更新提供了更高的吞吐量，并能很容易地处理不同的负载

同步更新和异步更新都是有用的。同步更新在事务系统中是很典型的，因为事务系统与用户交互，并需要协调用户接口。面向分析的工作负载或不需要协调的工作负载经常使用异步更新。异步更新的架构优势——更好的吞吐量和对负载高峰更好的管理——建议实现异步更新，除非你有不这样做的很好的理由。

12.5 过期实时视图

增量算法和随机写数据库使得速度层比批处理层和服务层复杂得多，但是 Lambda 架构最关键的一个优势是速度层的瞬态特性。因为更简单的批处理层和服务层不断覆盖速度层，所以速度层视图只需要表示还没有被批量计算 workflows 所处理的数据。一旦批量计算运行结

束，你就可以丢弃部分速度层视图——被服务层吸收的部分——但显然你必须保留其他所有数据。

在理想情况下，速度层数据库将提供直接对记录过期的支持，但这通常不是现有数据库的一个选项。Memcached 等工具具有一个类似的功能，可用于设定键/值对的延时到期时间，但它们并不适用于该问题。例如，你可以将记录的过期时间设置为它被获取到服务层视图之前的预期时间（包括可能作为缓冲的额外时间）。但如果由于意想不到的原因，批处理层的处理需要额外的时间，那么这些速度层的记录将提前过期。

作为替代，无论使用什么样的速度层数据库，我们都将就速度层视图过期提出一种通用的方法。为了理解这种方法，首先你必须了解每次服务层更新后需要将什么置为过期。前提是假设你有一个完整的 Lambda 架构实现，并且是第一次打开应用程序。系统还没有任何数据，所以最初的速度层视图和服务层视图都是空的。

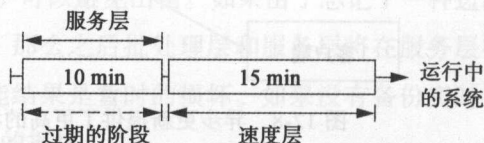
当批处理层第一次运行时，它将在空数据上进行操作。假设由于运行作业、创建空的索引等的开销，批处理层计算需要 10min。在 10min 结束时，服务层视图仍为空，但速度层视图此时表示了 10min 的数据。该情况如图 12-9 所示。

第二次运行批处理层时立即开始处理第一次运行期间所积累的 10min 的数据。为了说明这个观点，假设第二次运行需要 15min。当运行完成时，服务层视图将表示第一次的 10min 的数据，而速度层视图将表示所有 25min 的数据。现在可以从速度层视图中将第一次的 10min 的数据置为过期，如图 12-10 所示。



在运行第一次批量计算后，服务层保持清空状态，速度层接着处理当前数据

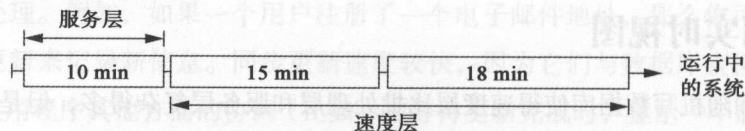
图 12-9 在第一次批量计算结束时服务层和速度层视图的状态



当第二次批量计算运行完成后，数据的第一部分已经传输到服务层，从速度层视图看是过期的

图 12-10 在第二次运行完成之后，可以将部分实时视图置为过期

最后，假设批处理层的第三次运行需要 18min。第三次批量计算运行完成之前的即时状态，如图 12-11 所示。



批处理层计算刚完成时，速度层负责处理之前两阶段运行累计的数据。

图 12-11 第三次批量计算运行完成之前，服务层和速度层视图的即时状态

此时，服务层仍然只表示了 10min 的数据，让速度层处理剩余的 33min 数据。该数字表明速度层视图弥补了批处理层第一次和第二次运行之间的时间，这取决于批处理层已经处理了多少的工作流。当该次批处理层运行结束时，这三次运行之前的数据可以安全地从速度层丢弃。

完成这项任务最简单的方法是，维护两组实时视图，并在每次批处理层运行之后交替清空它们，如图 12-12 所示。其中一组实时视图将准确地表示弥补服务层视图所必需的数据。在每次批处理层运行之后，应用程序应该将读取操作转换到有更多数据的实时视图（并离开刚被清空的视图）。

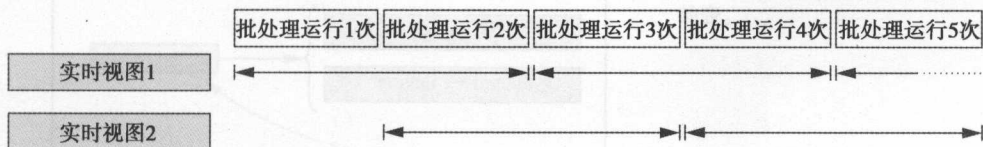


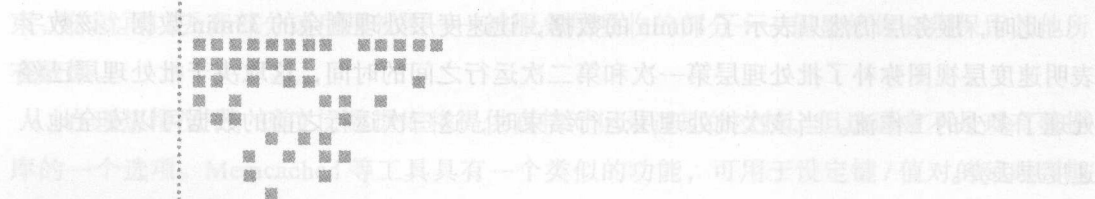
图 12-12 交替清空两组不同的实时视图，保证一组视图总是包含速度层的适当数据

乍一看，维护两个实时视图似乎是昂贵的，因为它其实使速度层的存储成本加倍了。关键是速度层视图只代表数据的很小一部分——最多几个小时的数据。与服务层表示的可能若干年的数据相比，这可能比系统所有数据的 0.1% 还要少。该方案的确引入了冗余，但为了得到过期实时视图的通用解决方案，这是可接受的代价。

12.6 总结

速度层与批处理层非常不同。你不是在整个数据集上计算函数，而是在更复杂的存储形式上用更复杂的增量算法进行计算。但 Lambda 架构允许你保持很小的速度层，因此它更易于管理。

你已经了解了速度层的基本概念和管理实时视图的细节，并看到了更新实时视图的两种方式——同步和异步。同步速度层没有什么更多的内容，因为你只是简单地连接到数据库并发起一个更新，但是异步速度层涉及了更多的内容。在探讨异步速度层之前，我们首先来看看 Cassandra——一个可以用于速度层视图的示例数据库。



实时视图：示例

作为替代，无论使用什么样的速度层数据库，我们都将就速度层（针对由实时数据源产生的数据）进行索引。假设你有一个完整的 Lambda 架构实现，并且是第一次打开应用程序。没有任何数据，所以最初的速度层视图和服务层视图都是空的。

当批处理层第一次运行时，它将在空数据上进行操作。假设由于运行作业、创建空的索引等的开销，批处理层计算需要 10min。在 10min 结束时，服务层视图仍为空，但速度层视图已包含数据。

本章内容

- Cassandra 的数据模型
- 使用 Cassandra 作为实时视图
- 通过控制分区和排序来支持广泛的视图类型

你已经了解了速度层和实时视图的基本概念，现在来看 Cassandra——一个可以用作实时视图的数据库。但无论如何，Cassandra 不是通用的实时视图——许多系统需要多个数据库来满足它们所有的索引、一致性和性能的需求。然而，本书之所以使用 Cassandra，是为了说明实时视图的概念。并且在本书中，Cassandra 将被用作 SuperWebAnalytics.com 速度层的数据库。由于很多公开、可用的资源可以帮你了解该数据库的内部运作，因此本章将从用户的角度关注 Cassandra 的属性。

13.1 Cassandra 的数据模型

尽管很多人说 Cassandra 是面向列的数据库，但我们发现，这种说法有些令人困惑。相反，将数据模型看作一个映射更简单，它将排序映射作为值（或者可以把已经带有排序映射的排序映射作为值）。Cassandra 允许基于嵌套映射的标准操作，比如添加键/值对、根据键查找和获得键的范围。

为了介绍术语，图 13-1 展示了 Cassandra 数据模型的不同方面。下面详述这些术语：

- ❑ **列族**——列族类似于关系数据库中的表，每个列族存储一组完整独立的信息。
- ❑ **键**——如果将列族看作一个巨大的映射，那么键是该映射中的顶级记录。Cassandra 使用键来跨集群对列族分区。
- ❑ **列**——每个键到另一个名称 / 值对的映射称为列。一个键的所有列被物理地存储在一起，这样使得访问若干范围的列不会很昂贵。不同的键可以有不同列的集合，并且对于一个给定的键，可以有几千甚至几百万的列。

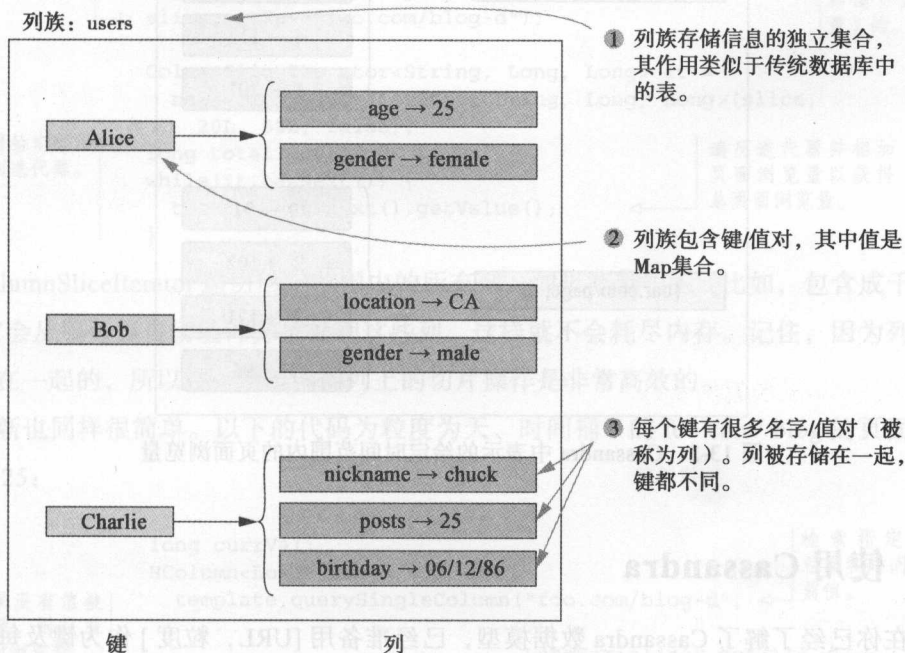


图 13-1 由列族、键和列组成的 Cassandra 数据模型

为了充分理解该模型，让我们回到 SuperWebAnalytics.com 的示例。具体地说，让我们看看如何使用 Cassandra 对给定时间范围内的页面浏览量建模。对于给定时间范围内的页面浏览量，你想要检索在特定时间桶范围内的一个特定 URL 和特定时间粒度（小时，天，周，或 4 周）的页面浏览量。为了将这些信息存储在 Cassandra 中，键将是 [URL, 粒度] 对，列是时间桶和页面浏览量的名称 / 值对。图 13-2 展示了该视图的一些样例数据。

这是存储页面浏览量的一种高效方式，因为列被存储——该示例下是根据时间桶来存储的——且被物理地存储在一起。

与上述方式形成鲜明对比的是，Cassandra 键是 URL、粒度和时间桶的三元组，并且列是单个页面浏览量的替代模式。在该模式中，查询一定范围的页面浏览量需要查找多个

Cassandra 的键。因为每个键可能存在于不同的服务器上，由于服务器响应时间的方差，这些查询可能会有高延迟，正如在第 7 章中所讨论的。

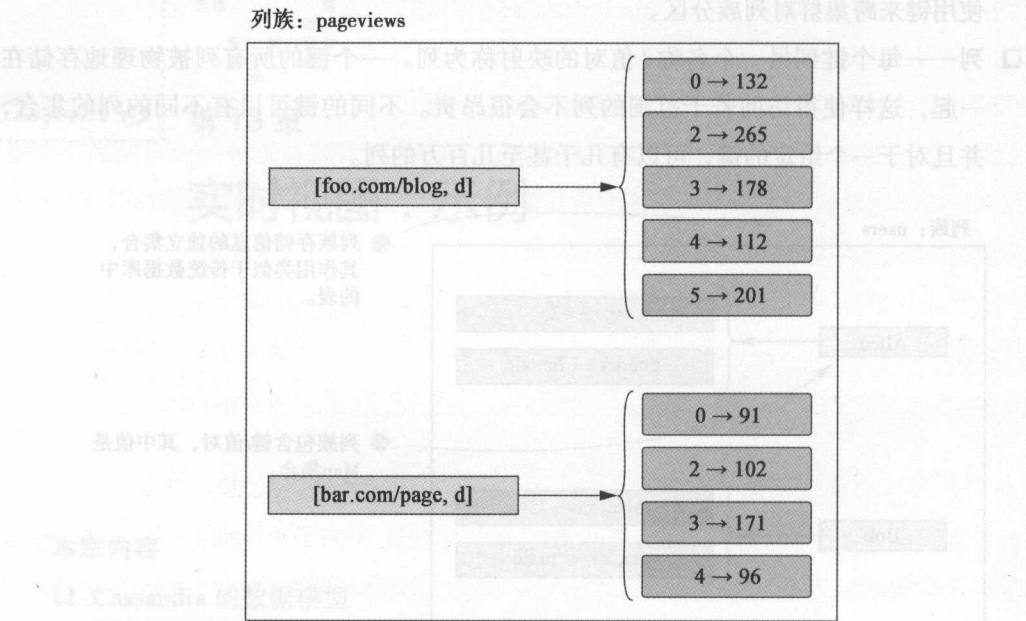


图 13-2 Cassandra 中表示的给定时间范围内的页面浏览量

13.2 使用 Cassandra

现在你已经了解了 Cassandra 数据模型，已经准备用 [URL, 粒度] 作为键及每个时间桶一个列，来实现给定时间范围内的页面浏览量模式。该实现将使用 Hector Java 客户端对 Cassandra 进行展示。

首先你需要一个能发起操作到指定列族的客户端。因为模式中的每个列族只需创建一次，所以此处略去模式定义代码以避免代码混乱。如果需要进一步地了解细节，参见 Hector 或 Cassandra 文档。

keyspace 是应用程序的所有列族的容器。

创建一个连接到分布式 Cassandra 集群的集群对象。

```
Cluster cluster = HFactory.getOrCreateCluster("mycluster", "127.0.0.1");  
→ Keyspace keyspace =  
    HFactory.createKeyspace("superwebanalytics", cluster);  
→ ColumnFamilyTemplate<String, Long> template =  
    new ThriftColumnFamilyTemplate<String, Long> (keyspace,  
    ColumnFamilyTemp-  
late 是用来发起操作到  
keyspace 的列族。    "pageviews",  
    StringSerializer.get(),  
    LongSerializer.get());
```

一旦有了与集群通信的客户端，你就可以检索给定 URL 和给定时间桶范围内的页面浏览量。下面的代码计算了以天为粒度的、20~55 的时间桶 foo.com/blog 的总页面浏览量：

```

// 为 Cassandra 查询创建一系列模板。
SliceQuery<String, Long, Long> slice =
    HFactory.createSliceQuery(keyspace,
        StringSerializer.get(),
        LongSerializer.get(),
        LongSerializer.get());
slice.setColumnFamily("pageviews");
slice.setKey("foo.com/blog-d");

ColumnSliceIterator<String, Long, Long> it =
    new ColumnSliceIterator<String, Long, Long>(slice,
        20L, 55L, false);
long total = 0;
while(it.hasNext()) {
    total += it.next().getValue();
}

```

为查询分配列族和键。

该序列化器是分别用于键（url/粒度），名字（桶）和值（页面浏览量）的。

获得给定范围的列迭代器。

遍历迭代器并相加页面浏览量以获得总页面浏览量。

ColumnSliceIterator 遍历给定范围内的所有列。如果范围很大（比如，包含成千上万的列），它会从服务器自动缓冲并批处理这些列，这样就不会耗尽内存。记住，因为列是排序并存储在一起的，所以在一定范围的列上的切片操作是非常高效的。

更新也同样很简单。以下的代码为粒度为天、时间桶 7 的 foo.com/ blog 的页面浏览量增加了 25：

```

long currVal;
HColumn<Long, Long> col =
    template.querySingleColumn("foo.com/blog-d",
        7L,
        LongSerializer.get());
if (col==null)
    currVal = 0;
else
    currVal = col.getValue();

ColumnFamilyUpdater<String, Long> updater =
    template.createUpdater("foo.com/blog-d");
updater.setLong(7L, currVal + 25L);
template.update(updater);

```

如果没有值被记录，则没有页面浏览量。

检索指定 URL、粒度和时间桶的当前值。

序列化页面浏览量值。

增量页面浏览量并执行更新。

为指定键创建一个更新模板。

对于像给定时间范围内的页面浏览量这样的分析示例，通常通过对多个时间桶进行批量更新来提高效率。在第 15 章讨论流处理时，我们将进一步探讨这个问题。

Cassandra 的高级特性

Cassandra 有一些高级特性值得一提，这使其适用于更广泛的实时视图类型。

Cassandra 的一个特性解决了 Cassandra 如何在集群的节点之间对键分区的问题。你可以在两种分区类型之间选择：RandomPartitioner 和 OrderPreservingPartitioner。

RandomPartitioner 使得 Cassandra 表现得像散列映射，并使用键的散列将键分配到分区。这会使键被均匀分布在集群的节点上。相比之下，OrderPreservingPartitioner 顺序地存储键，使得 Cassandra 表现得像排序映射。保持键的排序使你能够在若干范围的键上做高效查询。

虽然保持键的排序是有优势的，但使用 OrderPreservingPartitioner 是有成本的。当保存键的顺序时，Cassandra 试图分散键，以使每个分区包含大约相同数量的键。不幸的是，没有好的算法来动态确定平衡的键的范围。在增量的设置中，集群可能会变得不平衡，比如一些服务器超载而其他服务器几乎没有数据。这是面对实时环境、为了避免批量计算而使用增量计算，所引入的复杂性的另一个示例。在批处理环境中，你事先知道所有键，所以可以把键均匀地分散在分区上，并将该操作作为计算的一部分。

Cassandra 的另一个特性称为组合列，它进一步扩展了排序映射的想法。组合列允许嵌套任意深度的映射——比如，你可以将索引建模为排序映射的排序映射的排序映射的映射。这为你选择索引模型提供了极大的灵活性。

13.3 总结

本章介绍了使用 Cassandra 的基本知识。Cassandra 还有其他本章没有提及的特性，比如支持最终一致性，因为 SuperWebAnalytics.com 的示例不需要这些特性，所以不再赘述。你可以通过查询其他的资源来更多地了解 Cassandra。

作为一个随机读/随机写的数据库，Cassandra 明显比服务层数据库更难操作。在线压缩和调整键范围的需要（如果使用 OrderPreservingPartitioner）会造成很大的困难。幸运的是，Lambda 架构中速度层的即时性很大程度地保护了你——如果有什么严重错误，你可以舍弃速度层并予以重建。

下一个步骤是连接生成的数据流和实时视图。第 14 章将介绍如何将像 Cassandra 这样的数据库和流处理引擎组合起来完成这一壮举。

队列和流处理

本章内容

- 单消费者队列和多消费者队列
- 一次一个的流处理
- 流处理的队列和工作节点方法的局限性

你已经了解了两种类型的速度层架构：同步和异步。对于同步架构，应用程序直接发送更新请求到数据库，并阻塞直到收到响应。这样的应用程序需要协调不同的任务，但本书并没有从架构的角度进行过多的讨论。相反，异步架构更新速度层数据库是独立于创建数据的应用程序的。你决定如何持久化和处理更新请求，将直接影响整个系统的可扩展性和容错性。

本章讨论了队列和流处理的基本知识以及异步架构的两个基础。你之前看到的批处理的关键是具备承受失败和必要时重试计算的能力。速度层也需满足相同的原则，因为容错性和重试在流处理系统中是至关重要的。像往常一样，速度层的增量特性使得处理更复杂，而且在设计应用程序时，你要做出更多的权衡。

在讨论完持久化队列的需要之后，我们将简要介绍一种最简单的流处理模式——一次一个处理。你将看到在这种处理中容错和重试是如何发生的。

14.1 队列

为了理解为什么异步系统中需要持久化队列，首先考虑一个没有持久化队列的架构。

在这样的系统中，事件将被提交给独立处理每个事件的工作节点（Worker）。这种方法如图 14-1 所示。

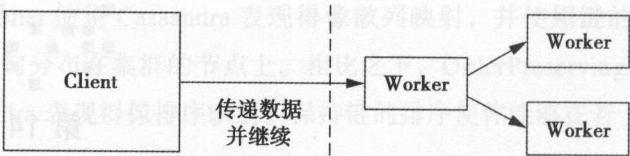


图 14-1 为了实现没有队列的异步处理，客户端提交一个事件，且并不监控它的处理是否成功

这个“即发即弃”方案不能保证所有数据被成功处理。例如，如果一個工作节点在完成分配的任务之前失败，那么并没有机制来检测或纠正该错误。该体系结构也容易在请求高峰时期失败，因为这超出了处理集群所拥有的资源。在这种情况下，集群将不堪重负，并且消息可能会丢失。

将事件写入持久队列解决了这两个问题。当工作进程失败时，队列允许系统重试事件，并且当下游工作节点遇到处理的限制时能够提供一个事件缓冲区。

虽然现在清楚了异步架构需要队列来持久化事件流，但要实现优良的队列语义设计，还需要进行进一步的讨论。最好从你已经熟悉的队列接口说起，如 Java 中的本地 Queue 接口。下面列出了接口中与此相关的方法：

从队列头移除记录。

```
interface Queue {  
    void add(Object item);  
    Object poll();  
    Object peek();  
}
```

将新记录添加到队列中。

←

在队列头检测记录而不移除。

←

14.1.1 单消费者队列

在设计速度层的持久化队列时，Java Queue 接口是一个固有的起点。事实上，如 Kestrel 和 RabbitMQ 的队列实现使用了类似的接口。它们共享相同的单消费者结构，代码如下：

检索用于处理的 Item。

```
struct Item {  
    long id;  
    byte[] item;  
}
```

下一个步骤是连接生成下一个 Item 的源。

←

一个通用的 Item 包含一个标识符和一个二进制负载。

←

当处理 Item 失败时报告。

```
interface Queue {  
    Item get();  
    void ack(long id);  
    void fail(long id);  
}
```

确认 Item 的成功处理。

←

单消费者队列的设计基于这样一种想法：当从队列中读取一个事件时，该事件不会立即被删除。相反，get 方法返回的记录包含一个标识符，稍后用它来确认处理事件成功 (ack) 或失败 (fail)。只有一个事件被确认成功，它才会被从队列中删除。如果事件处理失败或发生超时，队列服务器将允许另一个客户端通过一个单独的 get 调用来检索相同的事件。因此，若使用该方法，一个事件可能会被处理多次 (例如，一个客户端处理事件，但在确认成功之前宕机)，但每个事件至少被处理一次是毋庸置疑的。

这一切看似很好，但实际上该队列设计中有一个很深的缺陷——如果多个应用程序要使用相同的流将会怎样？这是很常见的。例如，给定一个页面浏览的流，一个应用程序可以建立给定时间范围内的页面浏览量视图，而另一个可以建立给定时间范围内的独立访客视图。

一个可能的解决方案是将所有应用程序封装在相同的消费者中，如图 14-2 所示。

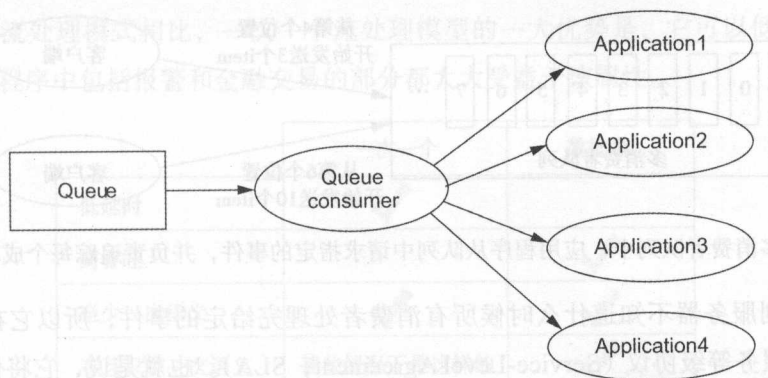


图 14-2 多个应用程序共享一个消费者队列

通过这种方式，所有应用程序驻留在相同的代码库，并运行在相同的工作节点中。但这是一个糟糕的设计，因为它消除了独立应用程序之间的任何隔离性。一旦没有隔离性，如果一个应用程序出现错误，它可能会影响在相同消费者中运行的所有其他应用程序。在较大的组织中多个组需要共享相同的流，该问题会更加严重。毫无疑问，隔离独立的应用程序会更好、更健全，这样，一个应用程序的损坏就不会波及其他应用程序。

对于单消费者队列的设计，实现应用程序之间独立的唯一方法是为每个消费者应用程序维护一个单独的队列。也就是说，如果有 3 个应用程序，那么你必须维护队列服务器上的 3 个独立的队列副本。这种方法最明显的缺点是大大增加了队列服务器上的负载。具体地说，现在的负载与应用程序数量和传入事件数量的乘积成正比，而不仅仅是传入事件的数量。你完全有可能使用相同的流建立几十个视图，而队列服务器在巨大负载下可能会宕机。

讨论单消费者队列的局限性有助于识别队列系统所需的属性。你真正想要的是一个可

以被多个消费者使用的队列，在该队列中增加消费者非常简单且只增加极小的负载。当你进一步思考这个问题时，单消费者队列的基本问题是，队列需要负责追踪所消费的事件。由于一个事件的限制条件是“被消费”或“没被消费”，因此队列无法优雅地处理多个客户端想要消费相同事件的情况。

14.1.2 多消费者队列

值得庆幸的是，还有一个可供选择的队列设计，它不受单消费者队列相关问题的影响。其想法是，将追踪事件的消费 / 未消费状态的责任从队列转移到应用程序本身。如果一个应用程序追踪所消费的事件，那么它可以请求从流历史中的任何点回放事件流。该设计如图 14-3 所示。

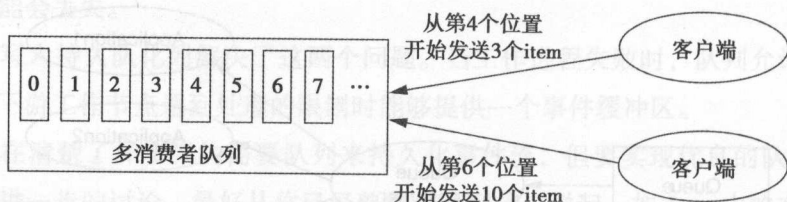


图 14-3 在多消费者队列中，应用程序从队列中请求指定的事件，并负责追踪每个成功处理的事件

因为队列服务器不知道什么时候所有消费者处理完给定的事件，所以它在可用事件上提供了一个服务等级协议 (Service-Level Agreement, SLA)。也就是说，它将保证有一定量的流是可用的，比如所有过去 12h 的事件或最后 50 GB 的事件。Apache Kafka 是多消费者队列实现的一个良好示例，它公开了一个类似于之前所描述的接口。

单消费者队列和多消费者队列还有一个显著的区别。对于单消费者队列，消息一旦被确认成功就被删除，并且不可以回放。因此，一个失败的事件可能导致事件流被乱序处理——如果流被并行地消费并且一个事件失败，在该失败事件之后的事件可能在重试失败事件之前被成功处理。相比之下，多消费者队列允许回放流和回放自故障点以来的任何事，确保能够按照事件的起始顺序来处理事件。回放事件流的能力是多消费者队列一个很大的优势，并且与单消费者队列相比，这些队列没有任何缺点。因此，我们强烈建议使用类似于 Apache Kafka 的多消费者队列。

14.2 流处理

一旦将传入事件提供给多消费者队列，下一个步骤就是处理这些事件并更新实时视图。

这种做法被称为流处理，如图 14-4 所示。

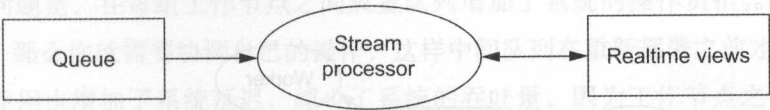


图 14-4 流处理

近年来出现了两类流处理的模型：一次一个流处理模型和微批量流处理模型。因为它们都有各自的优点和缺点，所以在使用时要进行取舍。它们是非常互补的——一些应用程序更适合一次一个流处理模型；而对于另外一些应用程序，微批量流处理模型是更好的选择。图 14-5 所示为两类模型的比较。本章着重介绍一次一个流处理模型，微批量流处理模型的相关内容将在第 16 章进行介绍。

与微批量流处理模式相比，一次一个流处理模型的一大优势是，它可以低延迟地处理流。示例应用程序中包括报警和金融交易的部分都大大受益于该属性。

	一次一个	微批量
低延时	✓	
高吞吐		✓
至少一次语义	✓	✓
有且只有一次语义	某些情况下是这样的	✓
简单编程模型	✓	

图 14-5 流处理模式的比较

我们将首先通过观察一种过时的方法，来开始构建一次一个流处理的通用模型——队列和工作者进程（Queues-workers）模型。困扰该方法的问题将催生一种更通用的方法，以完成这项任务。

14.2.1 队列和工作节点

队列和工作节点模式是实现一次一个流处理的一种常见方式。其基本思想是：将处理管道分到工作节点中，并在它们之间放置队列。使用该结构，如果一个工作节点失败或重新启动，它可以通过读取它的队列从上次中断处继续，如图 14-6 所示。

例如，假设使用队列和工作节点方法实现给定时间范围内的页面浏览量视图，如图 14-7 所示。第一组工作节点从一组队列中读取页面浏览事件，验证每个页面浏览来过滤

掉无效的 URL，然后将事件传递给第二组工作节点。然后第二组工作节点更新有效 URL 的页面浏览视图。

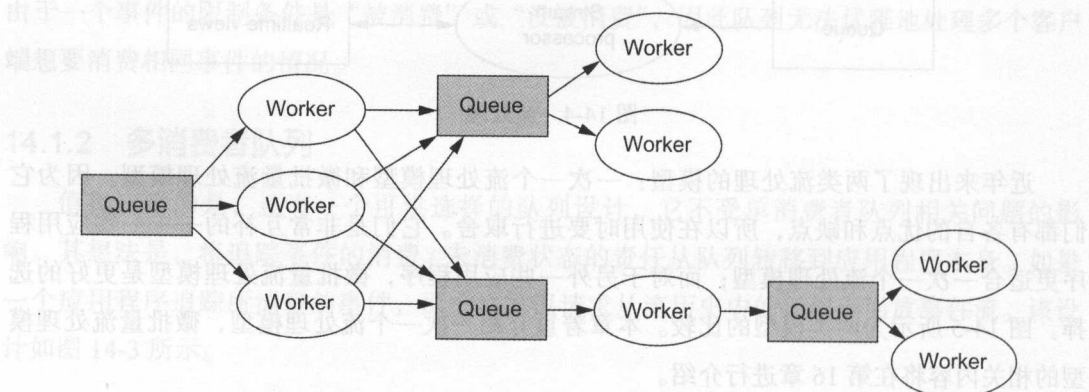


图 14-6 使用队列和工作节点架构的典型系统。图中的队列也可以是分布式队列

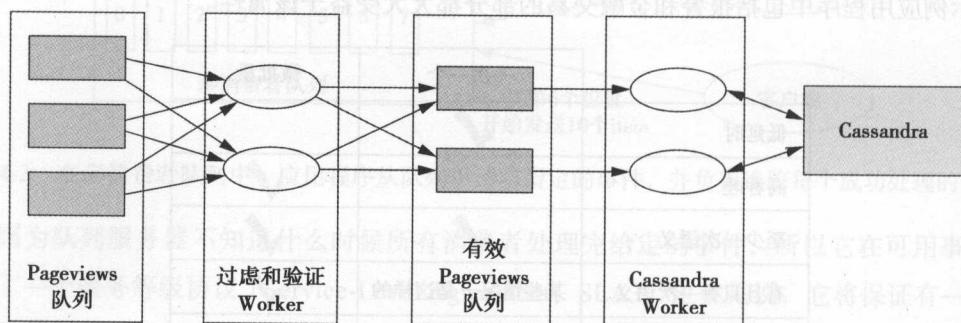


图 14-7 使用队列和工作节点架构计算给定时间范围内的页面浏览量

14.2.2 队列和工作节点的缺陷

队列和工作节点模式易懂但不一定简单。

该方式的微妙之处在于，确保多个工作节点不同时尝试更新相同 URL 的页面浏览计数。否则，可能会有潜在的竞态条件，使得数据库中的写操作会被损坏。为了满足这种要求，第一组工作进程根据 URL 来分区它的输出流。使用分区，整组的 URL 将被分散在队列中，但对于任何给定的 URL，页面浏览事件总是去到相同的队列中。实现这个目标的一种简单方法是，通过散列 URL 队列和根据目标队列的数量进行散列取模来选择目标队列。不幸的是，对队列进行分区的一个后果就是较差的容错性。如果一个更新数据库中页面浏览计数的工作节点失败，那么没有其他执行进程会更新数据库中的该部分流。这意味着必须在其他地方手动启动失败的工作节点，或构建一个自定义系统来自动地完成这个

工作。

另一个问题是，在每组工作节点之间放置队列增加了系统的操作负担。如果需要改变处理的拓扑，那么你就需要协调自己的操作，这样中间队列在重新部署之前才能被清除。

队列的使用也增加了系统延迟、减少了系统的吞吐量，因为工作节点之间每个事件的传递必须通过第三方，所以第三方中的每个事件必须被持久化到磁盘上。

除此之外，每个中间队列需要被管理和监控，并且添加了需要扩展的一层。

也许构建冗长乏味是队列和工作节点方法最大的问题。大部分代码将致力于完成序列化和反序列化通过队列传递的对象、连接工作节点池的路由逻辑和在服务器集群上部署工作节点的指令。当所有这些完成时，实际业务逻辑将最终只在代码库中占非常小的比例。

工作节点朝着更高的目标协力合作，同时需要非常细致的协调，这表明十分需要一个更高层次的抽象。

14.3 更高层次的一次一个的流处理

更高层次的一次一个的流处理方法是队列和工作节点模型的一种普通模型，且不包含任何复杂性。与队列和工作节点一样，该方案以一次一个元组的方式来处理元组，但代码跨集群并行地运行，所以系统是可扩展和高吞吐量的。记住，速度层的目标是处理流和更新实时视图——这就是它所做的全部工作。你的目标是以最少的开销和高可靠来保证、来完成这些数据处理工作。

从本书前面的章节可知，MapReduce 作为可扩展的批量计算模型，是 Hadoop 中一个特定的实现。与此类似，也有一个模型是为一次一个流处理而存在的，但是它没有简短、引人注意的名字。为了便于讨论，我们将把它称为 **Storm 模型**——这源于发起这些技术的项目。现在开始介绍这个模型，看它如何降低队列和工作节点的复杂性。

14.3.1 Storm 模型

Storm 模型将整个流处理管道表示为计算的图被称为**拓扑**。在队列和工作节点方式中，你需要为每个拓扑的节点编写单独的程序并手动连接它们，而 Storm 模型是将单个程序部署在集群上。这种灵活的方式使得单个可执行文件在第一个节点过滤数据，在第二个节点计算总量，在第三个节点更新实时视图数据库。序列化、消息传递、任务发现和容错可以交给抽象处理，这些都可以在实现低延迟的同时来完成（10 ms 或更少）。而在此之前，你

必须显式地对每个功能进行设计和编程。现在，你可以专注于业务逻辑了。

让我们从头开始构建 Storm 模型。Storm 模型的核心是流。如图 14-8 所示，流是无限的元组序列，元组是值的命名列表。本质上，Storm 模型是将流转换为新的流，可能会顺便更新数据库。

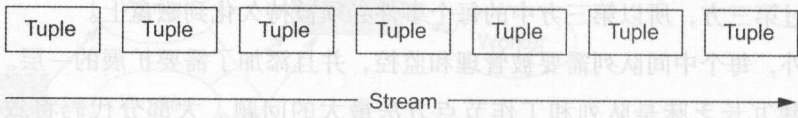


图 14-8 流是无限的元组序列

Storm 模型中的下一个抽象 Spout。Spout 是拓扑结构中流的来源（见图 14-9）。例如，Spout 可以从 Kestrel 或 Kafka 队列进行读取，并将数据转换成元组流，计时器 Spout 可以每 10s 发出一个元组到输出流中。

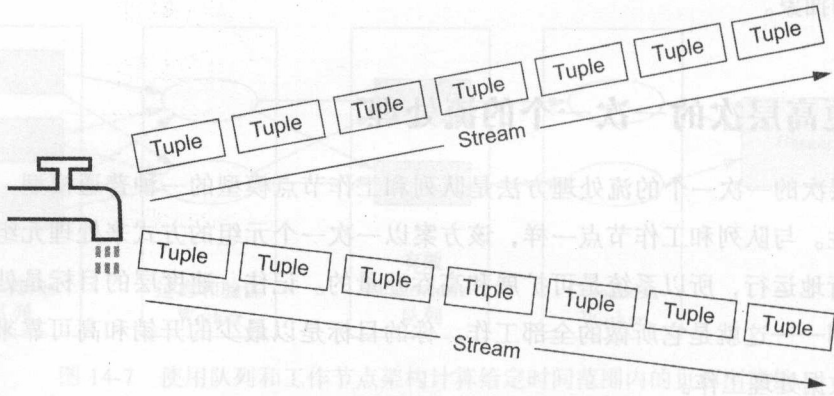


图 14-9 Spout 是拓扑结构中流的来源

Spout 是流的来源，而 Bolt 抽象执行流上的操作。Bolt 需要以任意数量的流作为输入，并生成任意数量的输出流（见图 14-10）。Bolt 实现拓扑结构中大部分的逻辑——它们运行函数、过滤数据、计算聚合、完成流连接、更新数据库等。

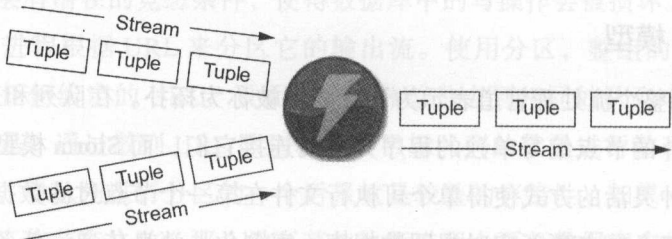


图 14-10 Bolt 处理来自一个或多个输入流的输入，并生成任意数量的输出流

定义完这些抽象，一个拓扑结构可以看作 Spout 和 Bolt 的网络，其中每条边代表一个处理另一个 Spout 或 Bolt 输出流的 Bolt（见图 14-11）。

每个 Spout 或 Bolt 的实例被称为任务。Storm 模型的关键在于任务本质上是并行的——就像 MapReduce 中的 map 和 reduce 任务本质上是平行的那样。流经拓扑的元组的并行性如图 14-12 所示。

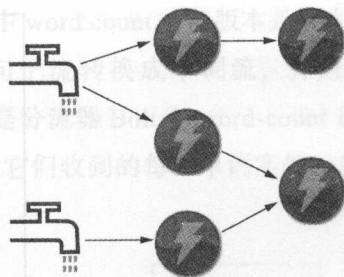


图 14-11 拓扑结构连接 Spout 和 Bolt，并定义元组如何流经一个 Storm 应用程序

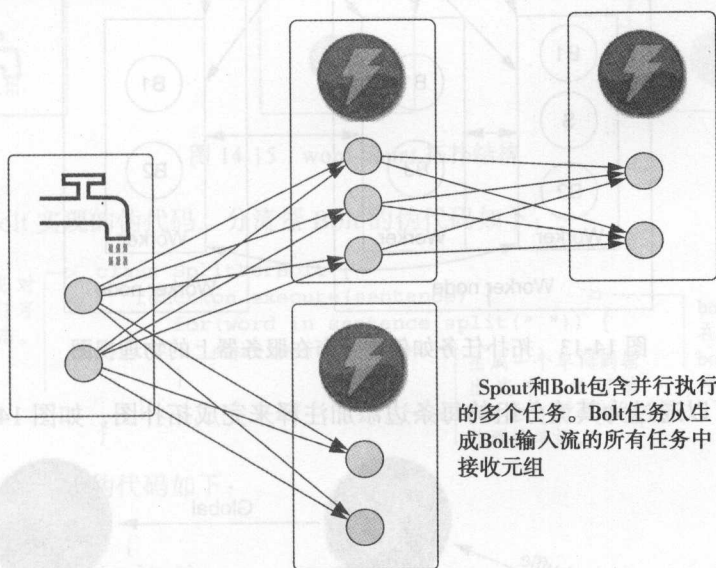


图 14-12 在一个拓扑结构中，Spout 和 Bolt 有多个实例在并行运行

当然，一个给定 Spout 或 Bolt 的所有任务不一定会运行在同一台机器上。相反，它们被分散在集群的不同工作节点中。图 14-13 描述了一个被物理机器所分组的拓扑结构。

Spout 和 Bolt 并行运行带来了一个关键的问题：当一个任务发出一个元组时，哪个消费任务应该接收它呢？Storm 模型需要流分组来指定应该如何分区元组到消费任务之间。最简单的一种流分组是随机分组，它使用随机循环算法来分布元组。该分组通过随机但均匀地将元组分配给所有消费者，来平均分散处理负载。另一种常见的分组是字段分组，它通过散列元组字段的一个子集，并使用消费任务的数量对该结果取模。例如，如果你在 word 字段上使用字段分组，那么所有相同单词的元组将被送到相同的任务中。

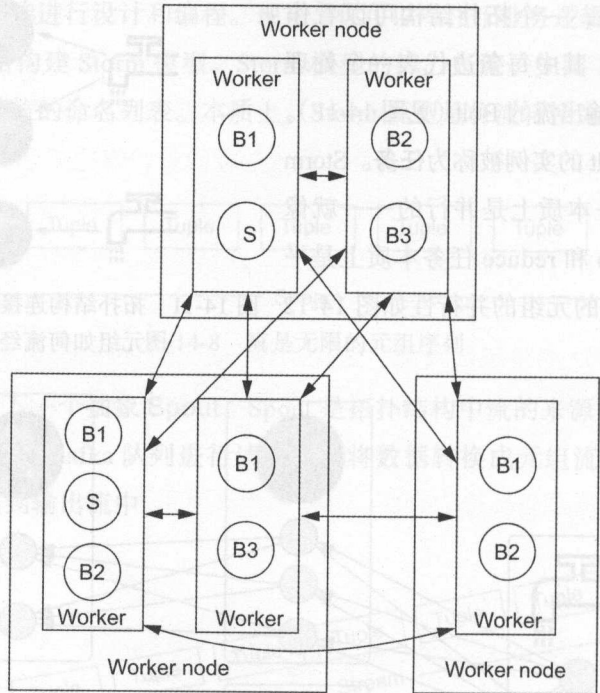


图 14-13 拓扑任务如何被分布在服务器上的物理视图

我们现在可以通过为其流分组的每条边添加注释来完成拓扑图，如图 14-14 所示。

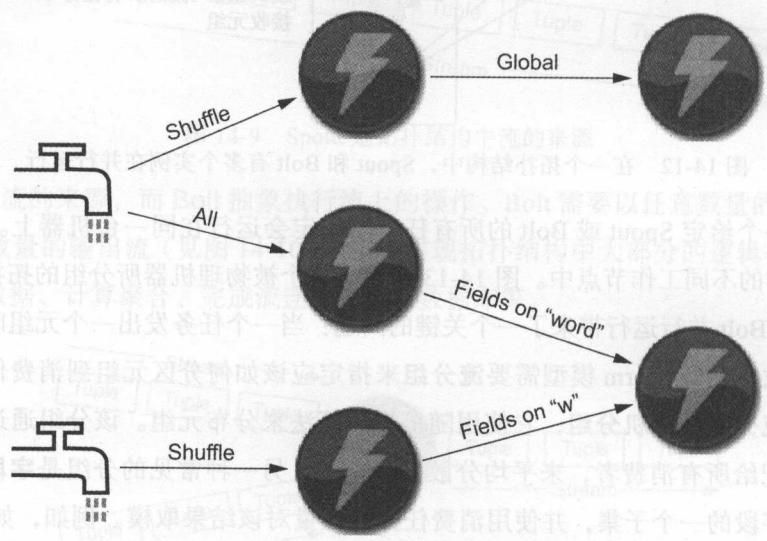


图 14-14 带有流分组的拓扑结构

让我们通过进一步深入一个基本的拓扑示例来巩固这个例子。正如 word count 是

MapReduce 示例中的关键引入，让我们看看 Storm 模型中 word count 的流版本是什么样的。

word-count 拓扑如图 14-15 所示。分流器 Bolt 将句子流转换成单词流，并且 word-count Bolt 消费这些单词来计算单词计数。这里的关键是分流器 Bolt 和 word-count Bolt 之间的字段分组。它确保了每个 word-count 任务可以看到它们收到的每个单词实例，从而计算出正确的计数。

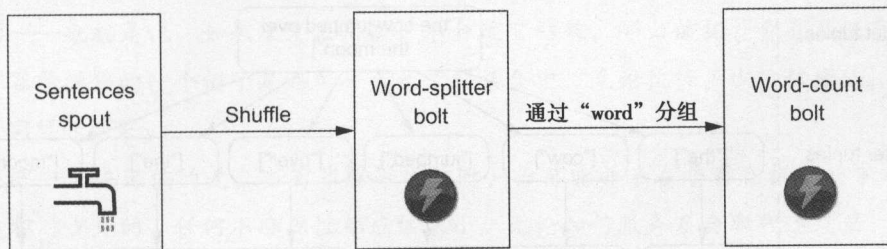


图 14-15 word-count 拓扑结构

现在来看 bolt 实现的伪代码。分流器 Bolt 的伪代码如下：

```

class SplitterBolt {
    function execute(sentence) {
        for(word in sentence.split(" ")) {
            emit(word)
        }
    }
}
  
```

← bolt 被定义成对象，因为它们可以保持内部状态。

← bolt 接收元组。在这种情况下，bolt 接收有一个字段的元组。

← 生成一个单词到输出流中。任何该 bolt 的订阅者将接收到该单词。

word-count Bolt 的伪代码如下：

```

class WordCountBolt {
    counts = Map(default=0)

    function execute(word) {
        counts[word]++
        emit(word, counts[word])
    }
}
  
```

← 单词计数被保存在内存中。

正如你看到的，Storm 模型不需要关于在哪里发送元组或如何序列化元组的逻辑。这些都可以在抽象之下被处理。

14.3.2 保证消息处理

在介绍队列和工作节点模型时，我们详细讨论了保存处理的每一阶段之间的中间队列的问题。Storm 模型的优势之一是，它不需要任何中间队列就可以实现。

如果有中间队列，消息处理会被保证，因为消息只有被工作进程成功处理，才会从队

列移除。如果工作进程死亡或有另一种失败，它将重试该消息，所以中间队列提供了“至少一次”处理的保证。

事实证明，即使没有中间队列，你也可以做到“至少一次”的保证。当然，工作机制必须不同——不是在失败发生的地方来发生重试，而是从拓扑的根处发生重试。为了理解这一点，让我们看看 word-count 拓扑中元组的处理是什么样的，如图 14-16 所示。

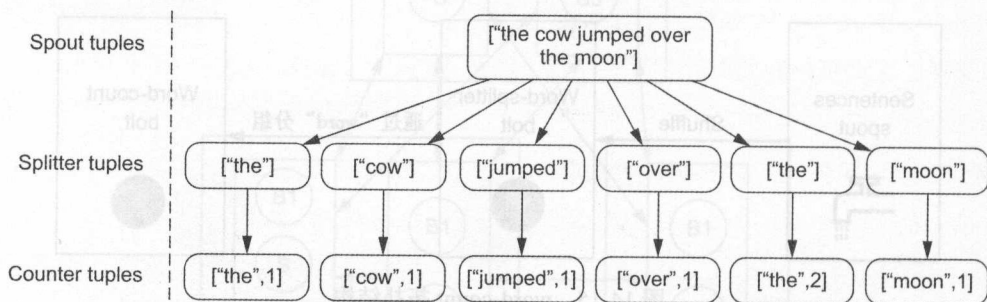


图 14-16 从 Spout 发出的单个元组的元组 DAG。DAG 的大小随着处理总量的增加而增长

当 Spout 生成句子元组时，句子元组被发送到任何订阅该 Spout 的 Bolt。在这种情况下，分词 Bolt 基于 Spout 元组创建 6 个新的元组。这些单词元组进入 word-count Bolt，该 Bolt 将为每个单词元组创建一个元组。你可以将单个 Spout 元组处理期间创建的每个元组看作一个有向无环图（DAG）。让我们称之为元组 DAG。可以想到，对于更复杂的拓扑，元组 DAG 将会更大。

可扩展和有效地追踪元组 DAG

你可能想知道如何可扩展和有效地追踪元组 DAG。元组 DAG 可能包含数百万个乃至更多的元组，你可能会直观地认为它需要大量的内存来追踪每个 Spout 元组的状态。事实证明，可以不通过显式追踪 DAG 来追踪元组 DAG——只需要为每个 Spout 元组维护 20B 的空间。无论 DAG 的大小如何，这都是正确的——即便 DAG 可能有几万亿个元组，20B 的空间仍然是足够的。这里不会讨论该算法（它被广泛记录在 Apache Storm 的在线文档中）。重要的是，该算法是非常有效的，并且这种有效性使其可以追踪失败并在流处理期间发起重试操作。

事实证明，有一个高效、可扩展的算法可以追踪元组 DAG，并且如果下游发生失败，可以重试来自 Spout 的元组。重试来自 Spout 的元组将导致整个元组 DAG 重新生成。

重试 Spout 似乎是后退的一步，因为这意味着已经完全成功的中间阶段将被再次尝试。但经进一步的检查，这实际上与之前并无不同。使用队列和工作节点，一个阶段可以被成

功处理，也可以在确认消息成功并从队列中删除之前失败，然后重试。在这两种场景中，处理仍然是“至少一次”保证。此外，当学习微批量流处理时，你将会看到，有且仅有一次语义可以通过在至少一次保证之上构建来实现，并且任何时候都不需要中间队列。

使用“至少一次”处理的保证

在很多情况下，重新处理元组将产生很少或不产生影响。如果一个拓扑中的操作是幂等的——也就是说，如果反复应用操作不会改变结果，那么该拓扑会有且仅有一次的语义。幂等操作的一个例子是添加一个元素到集合中，无论执行多少次该操作，得到的仍将是同样的结果。

还有一点要记住的是，当有非幂等操作时，你可能并不在乎存在极小的不准确性。失败是相对少见的，任何不准确性都应该很小。无论如何服务层会取代速度层，任何不准确性最终会被自动修正。此外，通过牺牲一些延迟可以实现有且仅有一次语义，但如果低延迟比暂时的不准确性更重要，那么在 Storm 模型中使用非幂等性操作是一个很好的折衷。

下面来看如何使用 Storm 模型实现 SuperWebAnalytics.com 速度层的一部分。

14.4 SuperWebAnalytics.com 速度层

回想要为 SuperWebAnalytics.com 实现的 3 个独立查询：

- ☐ 一段时间范围内的页面浏览量
- ☐ 一段时间范围的独立访客数据
- ☐ 一个域的跳出率

本节将实现独立访客查询，剩下的两个独立查询将在第 16 章实现。

该查询的目的是能够获得一段时间范围内一个 URL 的独立访客数量。回想一下，当为批处理层和服务层实现该查询时，出于效率使用了 HyperLogLog 算法——HyperLogLog 生成能与其他集合合并的压缩集合表示，从而无须存储每小时访客的集合，就能够计算一段时间范围的独立访客数量。需要进行取舍权衡的是，HyperLogLog 是一种近似算法，因此计数将有一小部分是不准确的。但是其节省的空间非常大，所以这是一个很容易做出的取舍，因为 SuperWebAnalytics.com 并不需要完美的准确度。相同的取舍权衡也存在于速度层，所以可以为给定时间范围内的独立访客的速度层版本使用 HyperLogLog。

如前所述，SuperWebAnalytics.com 可以使用 IP 地址和账户登录信息来追踪访客。如果

一个用户已登录，并在几乎同一时间使用手机和计算机访问相同的 Web 页面，那么用户的行为应该被记录为一个单次的访问。在批处理层中，通过使用等效边关系来追踪哪些标识符代表同一个人，然后规范化一个人的所有标识符到一个标识符，来实现这个目标。具体来说，我们可以在给定时间范围内的独立访客计算开始之前，执行一次完整的等效边分析。

在速度层中处理多个标识符的问题要复杂得多。这是因为多个标识符的关系可能在速度层视图更新之后才确定。例如，考虑以下的事件序列：

- 1) IP 地址 11.11.11.111 在下午 1:30 访问了 foo.com/about。
- 2) 用户 sally 在下午 1:40 访问了 foo.com/about。
- 3) 下午 2:00 发现了 11.11.11.111 和 sally 之间的等效边。

在应用程序了解该等价关系之前，这两次访问将归于两个不同的个体。为了尽可能得到最准确的统计计数，速度层必须对 [URL, hour] 独立访客计数减少 1。

让我们考虑在实时环境下这需要怎么做。首先，你需要实时追踪等效图，这意味着必须增量完成第 8 章中的整个图分析。其次，你必须能够确定相同的访客是否被多次计算。这就要求存储每小时访客的完整集合。HyperLogLog 是一种压缩的表示，无法帮助完成这项工作，所以处理等量问题首先排除了利用 HyperLogLog 的优势。除此之外，使用增量算法来完成等效图分析和调整之前计算的独立访客计数是相当复杂的。

不要求致力于完美地计算给定时间范围内的独立访客计数，你可以舍弃一些精度。记住，Lambda 架构的优势之一是，速度层中的精度舍弃并不是永久的精度舍弃——因为服务层持续覆盖速度层，任何不准确性都会被纠正，并且系统最终会展示准确性。因此你可以考虑替代的方法，并在它们引入的不准确性以及计算与复杂性的优势之间做权衡。

第一种替代方法是在实时环境下不执行等量分析。相反地，批量的等量分析结果可以通过键/值服务层的数据库创建并使其对速度层可用。在速度层中，在给定时间范围内的独立访客计算之前，PersonID 首先通过该数据库进行规范化。这种方法的优势在于，你可以利用 HyperLogLog 的优势而不必处理实时等量。这也大大简化了实现，并且是更少的资源密集型的。

现在让我们考虑这种方法在哪些地方是不准确的。因为预先计算等量分析是过时几个小时的，任何新发现的等量关系都没被利用到，因此，当用户导航到一个页面，注册了 UserID（这时捕捉到一条等量边），然后在同一个小时但从不同的 IP 地址返回到同一页面，这种情况下该策略是不准确的。**注意：**不准确性只发生在全新用户上——在用户注册已经被批处理等量分析处理之后，任何该用户的后续访问被正确记录。总体来说，对于节省的大量空间来说，只需舍弃很少量的不准确性。

你可能会舍弃其他准确性。第二种替代方法是完全忽略等量，并且只基于页面浏览中

的 PersonID 计算实时独立访客。在这种情况下，即使一个 UserID 和一个 IP 地址之间在几个月前已经被记录为等量，如果那个人访问了一个页面，并进行登录，然后再次访问了相同的页面，在该小时内，将为该独立访客计数记录多次。

正确的做法是运行批量分析来量化每种方法所产生的不准确性，这样就可以对采取哪种策略做出明智的决策。就直觉而言，似乎在速度层完全忽视等量不会引入太多的不准确性，所以为了保持这个例子的简单性，我们就演示这个方法。在继续之前，我们再次强调，任何速度层中的不准确性都是暂时的——整个系统作为一个整体最终是准确的。

拓扑结构

现在让我们忽视等量，来设计给定时间范围内的独立访客数量的速度层。具体步骤如下：

- 1) 使用包含用户标识符、URL 和时间戳的页面浏览事件的流。
- 2) 规范化 URL。
- 3) 更新包含从 URL 到小时到 HyperLogLog 集的嵌套映射的数据库。

图 14-17 所示为一个实现该方法的拓扑结构。

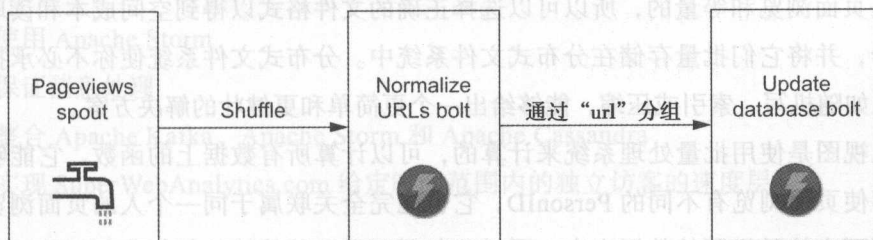


图 14-17 给定时间范围内的独立访客的拓扑结构

让我们更详细地观察每一步骤：

- ❑ 页面浏览 Spout——该 Spout 从队列进行读取并在页面浏览事件到达时发出它们。
- ❑ 规范化 URL Bolt——该 Bolt 规范化 URL 到它们的规范形式。你想要该规范化算法与批处理层中使用的一样，因此该算法在两层之间是一个共享库，这是很有意义的。此外，该 Bolt 可以过滤掉任何无效的 URL。
- ❑ 更新数据库 Bolt——该 Bolt 通过对 URL 使用字段分组，来消费前面的 Bolt 流，以确保在更新任何 URL 状态时没有竞态条件。该 Bolt 维护了数据库中的 HyperLogLog 集，它实现了 key-to-sorted-map 数据结构。键是 URL，嵌套的键是小时桶，嵌套的值是 HyperLogLog 集。在理想情况下，数据库将支持本地 HyperLogLog 集，以避免不得不从数据库中检索 HyperLogLog 集然后再将它们写回。

综上所述,通过忽视等量在速度层做出一个近似算法,速度层的逻辑被大大简化。

有必要强调一下,这样一个激进的近似算法可以在速度层实现,只是因为具备鲁棒性的批处理层支持它。在第10章中,你看到了给定时间范围内的独立访客的完全增量解决方案存在的问题,并且知道将等量添加到混合中使得一切非常困难。完全增量解决方案并没有忽视等量的选项,因为这将意味着忽视整个视图的等量。正如你刚才看到的,Lambda架构有更多的灵活性。

14.5 总结

你已经看到了增量处理具有非常紧密的延迟约束,其本质上比批量处理复杂得多。这是由于无法一次性查看所有数据和随机写操作数据库固有的复杂性(如压缩)造成的。然而大多数传统架构使用单个数据库来表示主数据集、历史索引和实时索引,这是复杂性的来源,因为所有这些事情应该更好地进行不同的优化,但不允许将它们交织到单一的系统。

SuperWebAnalytics.com 给定时间范围内的独立访客查询完美地展示了这三个步骤。主数据集是页面浏览和等量的,所以可以选择正确的文件格式以得到空间成本和读取成本的正确组合,并将它们批量存储在分布式文件系统中。分布式文件系统使你不必承担不必要的功能,如随机写、索引或压缩,能够给出一个更简单和更健壮解决方案。

历史视图是使用批量处理系统来计算的,可以计算所有数据上的函数。它能够分析等量图,即使页面浏览有不同的 PersonID,它也能完全关联属于同一个人的页面浏览。该视图被放到不支持随机写的数据库中,同时避免了不需要的特性的任何非必要的复杂性。因为在读取数据库时,数据库不会写入,所以你不必担心在线压缩等过程的操作负担。

最后,实时视图所需的关键属性是效率和低更新延迟。速度层通过增量地计算实时视图来实现这一点,通过忽视等量做出一个近似算法,这使得实现起来快速且简单。随机写数据库是用于实现速度层所需的低延迟,但它们的复杂性负担被实时视图本身就很小这一事实所大大抵消——大多数数据被批处理视图所表示。

在第15章中,你将了解如何使用现实生活中的工具来实现队列和流处理的概念。

队列和流处理：示例

本章内容

- 使用 Apache Storm
- 保证消息处理
- 整合 Apache Kafka、Apache Storm 和 Apache Cassandra
- 实现 SuperWebAnalytics.com 给定时间范围内的独立访客的速度层

在第 14 章中，你了解了多消费者队列和 Storm 模型作为一次一个流处理的通用方法。现在来看如何在实践中使用现实世界的工具 Apache Storm 和 Apache Kafka 来实现这些想法。本章最后将实现 SuperWebAnalytics.com 给定时间范围内的独立访客的速度层。

15.1 使用 Apache Storm 定义拓扑结构

Apache Storm 是一个开源项目，它实现了（并来源于）Storm 模型。如前所述，Storm 模型的核心概念是元组、流、Spout、Bolt 以及拓扑结构。下面使用 Apache Storm API 来实现流 word count。这里再次以 word-count 拓扑结构作为参考，如图 15-1 所示。

首先通过实例化一个 TopologyBuilder 对象来定义应用程序拓扑结构。TopologyBuilder 对象公开了用于指定 Storm 拓扑的 API：

```
TopologyBuilder builder = new TopologyBuilder();
```

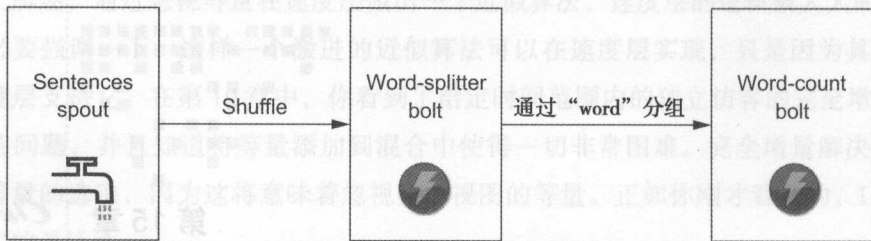


图 15-1 word-count 拓扑结构

接下来，添加一个 Spout 并产生句子流。该 Spout 被命名为 sentence-spout，并被指定并行度为 8，这意味着有 8 个线程将跨集群执行该 Spout：

```
builder.setSpout("sentence-spout", new RandomSentenceSpout(), 8);
```

现在已经有有了一个句子流，那么就需要一个 Bolt 来使用该流并将其转换成单词流。该 Bolt 被称作分流器 (Splitter)，且并行度为 12。因为没有要求如何消费句子，所以使用随机分组来均匀地在所有 12 个任务上分布处理负载：

```
builder.setBolt("splitter", new SplitSentence(), 12)
    .shuffleGrouping("sentence-spout");
```

最后一个 Bolt 使用单词流并产生所需单词计数的流。它被形象地称为计数器 (Counter)，且并行度也为 12。需要注意的是，应使用字段分组，以确保只有一个任务负责确定任何特定单词的总数：

```
builder.setBolt("count", new WordCount(), 12)
    .fieldsGrouping("splitter", new Fields("word"));
```

至此，拓扑已经定义完毕，你可以继续 Spout 和 Bolt 的实际实现。分流器 Bolt 的实现非常简单。它提取输入元组第一个字段的句子，并为句子中的每个单词发出一个新的元组：

```
public static class SplitSentence extends BaseBasicBolt {
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String sentence = tuple.getString(0);
        for (String word: sentence.split(" ")) {
            collector.emit(new Values(word));
        }
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

分割句子并发出每个单词到输出元组流

一个输入元组包含一个句子。

声明输出元组包含标签为“word”的单个值

计数器 bolt 的逻辑也很简单。这种特定的实现使单词计数是一个保持在内存中的

hashmap，但你可以很容易地使其与数据库通信。

```

public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts =
        new HashMap<String, Integer>();

    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if(count==null) count = 0;
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));

        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word", "count"));
        }
    }
}

```

从输入元组中提取单词。

一个内存映射存储该 bolt 接收的所有单词的计数。

检索当前单词的计数。

存储更新后的计数。

为每个给定的单词发出更新后的计数。

声明输出元组包含单词及其当前计数。

如果该单词之前没有被观察到，则初始化计数。

剩下的工作就是 Spout 实现。Storm 提供了许多预先构建的 Spout，类似于 Kafka 或 Kestrel，用于从外部队列读取数据，但下面的代码展示了如何构建一个自定义的 Spout。该 Spout 每 100 ms 随机发出它的一个句子，创建句子的一个无限流：

```

public static class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    public void open(Map conf, TopologyContext context,
        SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();

        public void nextTuple() {
            Utils.sleep(100);

            String[] sentences = new String[] {
                "the cow jumped over the moon",
                "an apple a day keeps the doctor away",
                "four score and seven years ago",
                "snow white and the seven dwarfs",
                "i am at two with nature";
            };
            String sentence = sentences[_rand.nextInt(sentences.length)];
            _collector.emit(new Values(sentence));

            public void declareOutputFields(OutputFieldsDeclarer declarer) {
                declarer.declare(new Fields("sentence"));
            }
        }
    }
}

```

Storm 循环调用 next-Tuple 方法。

使当前线程休眠 100 ms。

该 Spout 发出的一个句子的数组。

随机发出其中的一个句子。

声明输出元组包含一个单独的句子。

这就是 Storm 的拓扑结构及其实现。下面来看 Storm 集群如何工作以及如何部署它们。

15.2 Apache Storm 集群及其部署

Storm 集群的体系结构如图 15-2 所示。Storm 有一个主节点称为 Nimbus，负责管理正在运行的拓扑。Nimbus 接收在 Storm 上部署拓扑的请求，并分配集群的 Worker 来执行该拓扑。Nimbus 还负责检测 Worker 什么时候宕机，必要时将它们重新分配到其他机器。

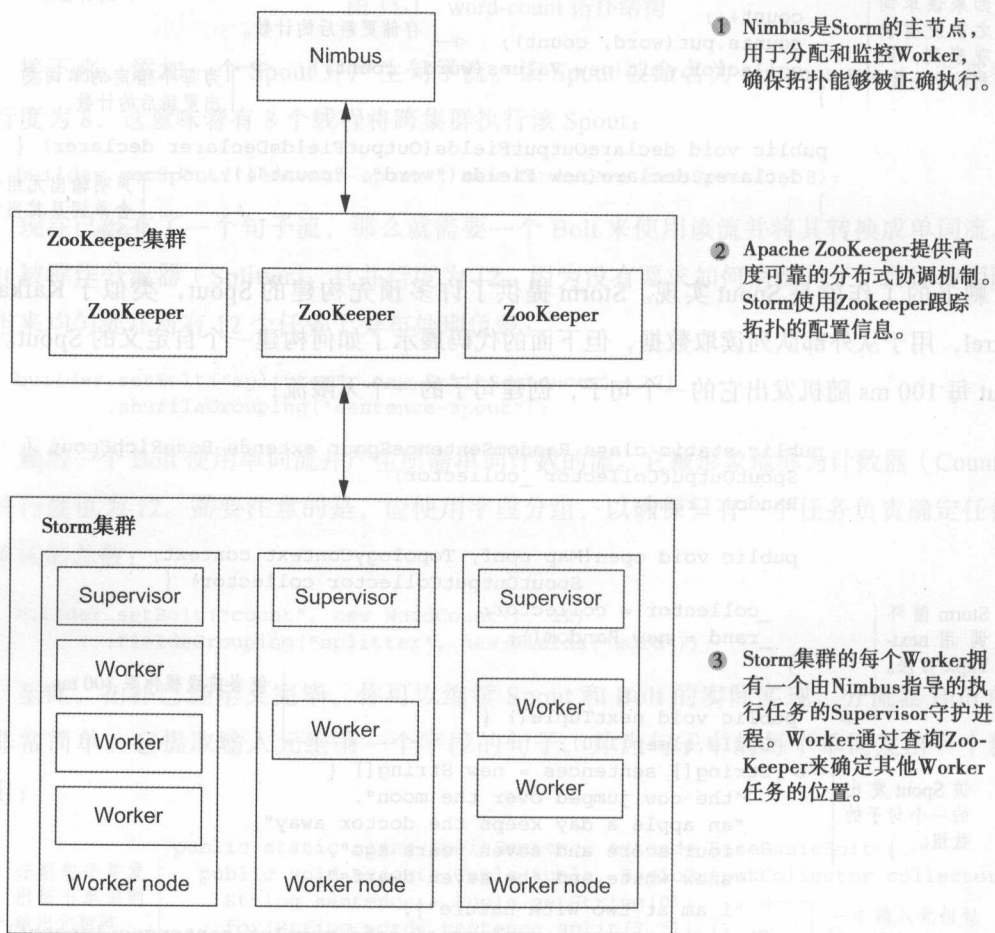


图 15-2 Apache Storm 体系结构

该结构图的中心是 ZooKeeper。ZooKeeper 是另一个 Apache 项目，擅长维护少量的状态并有适合集群协调的优良语义。在 Storm 体系结构中，ZooKeeper 追踪 Worker 的分配位置和其他拓扑的配置信息。在 Storm 中，典型的 ZooKeeper 集群是 3 个或 5 个节点。

Storm 集群中的最后一组节点由 Worker 节点组成。每个 Worker 节点运行一个被称为 Supervisor 的守护进程，该守护进程通过 ZooKeeper 与 Nimbus 通信来确定应该在机器上运行什么。然后在必要的时候，Supervisor 启动或停止 Worker 进程，就像 Nimbus 直接操作的一样。一旦运行，Worker 进程将通过 ZooKeeper 发现其他 Worker 的位置，并直接互相传递信息。

下面来看如何部署 15.1 节中构造的 word-count 拓扑结构：

```
public static void main(String[] args) throws Exception {
    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("sentence-spout", new RandomSentenceSpout(), 8);
    builder.setBolt("splitter", new SplitSentence(), 12)
        .shuffleGrouping("sentence-spout");

    builder.setBolt("count", new WordCount(), 12)
        .fieldsGrouping("splitter", new Fields("word"));
```

```
    Config conf = new Config();
    conf.setNumWorkers(4);
    StormSubmitter.submitTopology(
        "word-count-topology",
        conf,
        builder.createTopology());
    conf.setMaxSpoutPending(1000);
}
```

当提交该拓扑时提供一个名字。

在 Storm 服务器之间生成 4 个 Worker 节点。

设置一个 Spout 可以发出的未被确认的元组的最大数量。

拓扑配置包含适用于整个拓扑的参数。在这段代码样例中，配置 Storm 在集群上生成 4 个 Worker 来执行拓扑。回想一下，当完成拓扑定义时，你指定了每个 Spout 和 Bolt 的并行度：句子 Spout 的并行度为 8，分流器 Bolt 和计数器 Bolt 的并行度都是 12。这些并行度的值指示了应该为 Spout 或 Bolt 生成的线程数量。因此，该拓扑需要 4 个 Java 进程来执行 32 个线程。在默认情况下，Storm 跨集群均匀地分布 Worker，并在 Worker 上均匀地分布任务，但是你可以通过将自定义调度器嵌入 Nimbus 来改变分配策略。

代码样例也有第二个拓扑范围的配置设置，该设置用于处理传入数据的激增。如果有激增输入事件，重要的是流处理器不会由于增加的负载（例如内存耗尽）而不堪重负进而导致。基于保证消息处理的特性，Storm 有一种简单的机制来管理流控制。拓扑 max spout pending 设置控制一个 Spout 可以发出的未被拓扑完全处理的最大元组数。一旦达到这一限制，Spout 任务将停止发出元组直到元组被确认、失败或超时。在前面的示例中，代码告知 Storm 任何一个 Spout 任务的待处理的元组最大数量是 1000。因为句子 Spout 的并行度是 8，那么整个拓扑中待处理的元组数量最多是 8000。

15.3 保证消息处理

如第 14 章中所述, 使用 Storm 模型无需中间消息队列就可以保证消息处理。当检测到元组 DAG 中 Spout 在下游失败时, Spout 将重试元组。下面讨论使用 Apache Storm 的细节。

只有当整个元组 DAG 执行完并且每个节点已经被标记为完成, Storm 才认为 Spout 元组被成功处理。此外, 整个过程需要在指定的超时内发生(默认 30s)。超时确保无论下游发生什么, 失败都将被检测到——无论 Worker 进程挂起或机器突然宕机。

作为用户, 为了利用消息处理保证机制, 你有两项职责: 当创建一条依赖边到元组 DAG 时, 你必须通知 Storm; 当处理元组完成时, 你也必须通知 Storm。这两个任务分别称为 **anchoring** 和 **acking**。流 word count 中的适当地使用元组 DAG 逻辑的 sentence-splitter 代码如下:

```
public static class SplitExplicit extends BaseRichBolt {
    OutputCollector _collector;

    public void prepare(Map conf, TopologyContext context,
                       OutputCollector collector) {
        _collector = collector;
    }

    public void execute(Tuple tuple) {
        String sentence = tuple.getString(0);
        for(String word: sentence.split(" ")) {
            _collector.emit(tuple, new Values(word));
        }
        _collector.ack(tuple);
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

BaseRichBolt 子类需要显式地处理元组的 anchoring 和 acking。

标记输出的单词元组到输入的句子元组

确认句子元组被成功处理

该 Bolt 的语义实际上与前一节中的原始实现是相同的。当一个新的单词元组被发出时, 句子元组作为第一个参数被包含在内。该过程标记单词元组到句子元组。在新的元组发出后, 句子元组被确认, 因为它不再需要进一步的处理。Bolt 标记所有输出元组到输入元组, 是一种非常常见的模式。为了自动化该行为, Storm 提供了 BaseBasicBolt 类, 负责处理这种类型的 anchoring/acking。分流器 Bolt 的第一种实现使用了 BaseBasicBolt。

但 BaseBasicBolt 模式并不适用于所有操作, 特别是聚合或连接流的操作。例如, 假设你想同时处理 100 个元组, 在这种情况下, 你可以在一个缓冲区中存储所有传入的元组, 标记输出元组到所有的 100 个元组, 然后在缓冲区中确认所有元组。下面的代码通过发出

每 100 个元组的总和展示了这种策略：

```

public static class MultiAnchorer extends BaseRichBolt {
    OutputCollector _collector;

    public void prepare(Map conf, TopologyContext context,
        OutputCollector collector) {
        _collector = collector;
    }

    List<Tuple> _buffer = new ArrayList<Tuple>();
    int _sum = 0;

    public void execute(Tuple tuple) {
        _sum += tuple.getInteger(0);
        if (_buffer.size() < 100) {
            _buffer.add(tuple);
        } else {
            _collector.emit(_buffer, new Values(_sum));
            for (Tuple _tuple : _buffer) {
                _collector.ack(_tuple);
            }
            _buffer.clear();
            _sum = 0;
        }
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("sum"));
    }
}

```

MultiAnchorer bolt 加和 100 个输入元组的值并发出总和。

为传入的值维护一个运行的总和。

添加到缓冲区直至达到容量。

在缓冲区中确认所有元组。

当缓冲区满时，发出标记到缓冲区中所有元组的带有总和的元组。

清空缓冲区。

重置运行的总和。

在这种情况下，不能使用 BaseBasicBolt，因为元组被 execute 方法处理之后没有被立即确认——它们被缓冲并稍后被确认。

在内部，Storm 使用一种高效的算法来追踪元组 DAG，对于每个 Spout 元组，只需要大约 20 B 的空间即可。无论 DAG 的大小如何，这都是成立的——即便它可能有几万亿个元组，20 B 的空间仍然是足够的。这里不会详细讨论该算法，它在 Storm 的网站上被广泛地记录。重要的是，该算法是非常高效的，并且这种有效性使其可以追踪失败并在流处理期间发起重试操作。

15.4 实现 SuperWebAnalytics.com 给定时间范围内的独立访客的速度层

第 14 章介绍了 SuperWebAnalytics.com 给定时间范围内的独立访客的速度层设计。其主要思想是通过忽视等量做出一个近似算法，从而大大简化实现。这里再次以该速度层的

拓扑结构作为参考，如图 15-3 所示。下面将使用 Apache Storm、Apache Kafka 和 Apache Cassandra 来实现该拓扑。

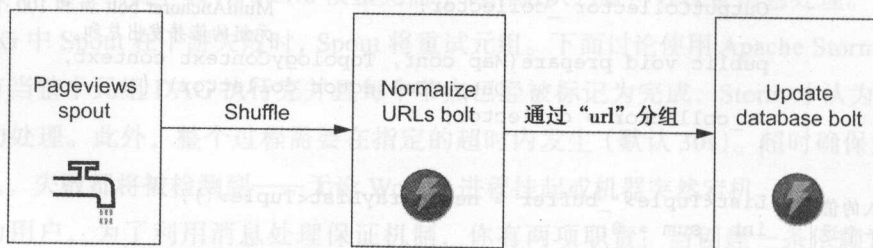


图 15-3 给定时间范围内的独立访客的拓扑结构

为了实现该拓扑，首先从 Spout 开始。下面的代码初始化一个 Kafka Spout，用于从 Kafka 服务器集群中读取页面浏览量。页面浏览量被认为以第 3 章中定义的 Thrift Data 对象，存储在 Kafka 上。

```

TopologyBuilder builder = new TopologyBuilder();
SpoutConfig spoutConfig = new SpoutConfig(
    new KafkaConfig.ZkHosts("zkserver:1234", "/kafka"),
    "pageviews",
    "/kafkastorm",
    "uniquespeedlayer");
spoutConfig.scheme = new PageviewScheme();
builder.setSpout("pageviews",
    new KafkaSpout(spoutConfig), 16);
  
```

该 Spout 的 Kafka 主题。 → 指向 `new KafkaConfig.ZkHosts("zkserver:1234", "/kafka")`

该 Spout 的 ID。 → 指向 `"pageviews"`

创建并行为 16 的 Spout。 → 指向 `new KafkaSpout(spoutConfig), 16`

ZooKeeper 集群的地址和在 ZooKeeper 内 Kafka 使用的命名空间。 → 指向 `new KafkaConfig.ZkHosts("zkserver:1234", "/kafka")`

Storm 将为该 Spout 使用的 ZooKeeper 命名空间。 → 指向 `"uniquespeedlayer"`

设置模式来说明 Spout 如何将二进制记录反序列化为 Data 对象。 → 指向 `spoutConfig.scheme = new PageviewScheme();`

其中大部分代码都是配置项设置：Kafka 集群的细节和使用的主题，以及 Spout 在 ZooKeeper 的何处记录到目前为止它所消费的东西。

下一步是规范化页面浏览事件中的 URL：

```

public static class NormalizeURLBolt extends BaseBasicBolt {
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        PersonID user = (PersonID) tuple.getValue(0);
        String url = tuple.getString(1);
        int timestamp = tuple.getInteger(2);

        try {
            collector.emit(new Values(user,
                normalizeURL(url),
                timestamp,
                user));
        } catch (MalformedURLException e) {}
    }
}
  
```

尝试使用应该与批处理层共享的函数来规范化 URL。 → 指向 `normalizeURL(url)`

如果规范化 URL 发生失败，那么通过不发出任何东西来过滤元组。 → 指向 `catch (MalformedURLException e) {}`


```

public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("user", "url", "timestamp"));
}
}

```

最后一步是更新存储在 Cassandra 中的 HyperLogLog 集。下面从一个简单的版本开始。这段代码检索相应页面浏览的 HyperLogLog 集并更新集合，然后将集合写回 Cassandra：

```

public static class UpdateCassandraBolt extends BaseBasicBolt {
    public static final int HOURS_SECS = 60 * 60;

    ColumnFamilyTemplate<String, Integer> _template;

    public void prepare(Map conf, TopologyContext context) {
        Cluster cluster =
            HFactory.getOrCreateCluster("mycluster", "127.0.0.1");

        Keyspace keyspace =
            HFactory.createKeyspace("superwebanalytics", cluster);

        _template =
            new ThriftColumnFamilyTemplate<String, Integer> (keyspace,
                "uniques", StringSerializer.get(), IntegerSerializer.get());
    }

    public void execute(Tuple tuple, BasicOutputCollector collector) {
        PersonID user = (PersonID) tuple.getValue(0);
        String url = tuple.getString(1);
        int bucket = tuple.getInteger(2) / HOURS_SECS;

        HColumn<Integer, byte[]> hcol =
            _template.querySingleColumn(url, bucket,
                ByteArraySerializer.get());

        HyperLogLog hll;
        try {
            if (hcol==null) hll = new HyperLogLog(800);
            else hll = HyperLogLog.Builder.build(hcol.getValue());

            hll.offer(user);
            ColumnFamilyUpdater<String, Integer> updater =
                _template.createUpdater(url);
            updater.setByteArray(bucket, hll.getBytes());
            _template.update(updater);
        } catch (IOException e) { throw new RuntimeException(e); }

        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            // empty since the bolt does not emit an output stream
        }
    }
}

```

初始化从 Cassandra 中检索和存储值的类。

从输入元组中提取用户、URL 和小时桶。

从 Cassandra 中检索 HyperLogLog 集，如果没有检索到，则初始化一个新的 HyperLogLog 集。

将用户添加到 HyperLogLog 集并更新 Cassandra。

出于完整性的考虑，使用下面的代码将该拓扑连接在一起：

```

public static void main(String[] args) {
    TopologyBuilder builder = new TopologyBuilder();
    SpoutConfig spoutConfig = new SpoutConfig(
        new KafkaConfig.ZkHosts("zkserver:1234", "/kafka"),
        "pageviews",
        "/kafkastorm",
        "uniquespeedlayer");

    spoutConfig.scheme = new PageviewScheme();
    builder.setSpout("pageviews",
        new KafkaSpout(spoutConfig), 16);

    builder.setBolt("extract-filter", new NormalizeURLBolt(), 32)
        .shuffleGrouping("pageviews");
    builder.setBolt("cassandra", new UpdateCassandraBolt(), 16)
        .fieldsGrouping("extract-filter", new Fields("url"));
}

```

注意：拓扑是完全容错的。因为 Spout 元组只有在数据库被更新之后才被认为是确认过的，所以任何失败都将导致 Spout 元组回放。失败和重试不会影响系统的准确性，因为增加到 HyperLogLog 集是一个幂等的操作。

上述 Cassandra 代码的问题是，从 Cassandra 检索集合并将它们写回需要大量的开销。理想的数据库应该支持本地 HyperLogLog，所以应不再需要这样的开销，但是 Cassandra 没有这个特性。

你仍然可以通过批量更新使事情更有效，尤其是对于相同的集合可以同时更新多次的情况。下面的代码展示了批量方法的一个模板，每几百个元组或每秒一次地写入 Cassandra，无论先满足哪个条件：

```

public static class UpdateCassandraBoltBatched extends BaseRichBolt {
    public static final int HOURS_SECS = 60 * 60;

    List<Tuple> _buffer = new ArrayList();
    OutputCollector _collector;

    public void prepare(Map conf, TopologyContext context,
        OutputCollector collector) {
        _collector = collector;
        // set up Cassandra client here
    }

    public Map getComponentConfiguration() {
        Config conf = new Config();
        conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, 1);
        return conf;
    }

    public void execute(Tuple tuple) {
        boolean flush = false;

```

每秒生成一个标志元组
来确保最少每秒一次的
更新。

```

if(tuple.getSourceStreamId()
    .equals(Constants.SYSTEM_TICK_STREAM_ID)) {
    flush = true;
} else {
    _buffer.add(tuple);
    if(_buffer.size() >= 100) flush = true;
}

if (flush) {
    // batch updates to Cassandra here
    for(Tuple t: _buffer) {
        _collector.ack(t);
    }
    _buffer.clear();
}

public void declareOutputFields(OutputFieldsDeclarer declarer) {
    // empty since the bolt does not emit an output stream
}

```

如果当前元组是标记元组，就更新。

如果是常规元组，将其添加到缓冲区并当缓冲区满的时候进行更新。

确认所有元组然后清空缓冲区。

这段代码的一个关键之处在于，元组被缓冲并于相应的更新在 Cassandra 中被批量处理后才被确认。这将确保如果有任何失败，那么可以进行回放。为了确保更新至少每秒发生一次，使用称为标记元组的 Storm 特性。标记元组配置为每秒一次的 Bolt。如果其中的标记元组传入，无论当前的缓冲区怎样，它都会被写入数据库。我们忽略了该代码中的 Cassandra 部分，因为它理解起来有些困难，并偏离了代码的流处理方面。

还有很多可以使该代码更加高效的方法。考虑下面可能的优化列表：

- ❑ 批量计算可以估计不同域的 HyperLogLog 集的所需大小（拥有更多独立访客的域需要更大的 HyperLogLog 集）。大多数域只需要非常小的 HyperLogLog 集，提前了解这些可能会节省相当大的存储空间。
- ❑ 你可以为 Storm 集群实现一个自定义调度器，这样 Cassandra Bolt 任务可以被配置到它们更新的 Cassandra 分区。这将消除更新任务和 Cassandra 之间的网络传输。
- ❑ 如前所述，如果 Cassandra 可以实现本地 HyperLogLog，那么 HyperLogLog 集不必来回地进行传输。

实现所有这些优化超出了这本书的范围——这些只是可以用于改进速度层的技术上的建议。

15.5 总结

现在你应该对速度层的所有概念有了很好的理解——队列、流处理器和实时视图。由

于速度层增量的特性，到目前为止，它是任何架构中最复杂的一部分，对比本章中的增量代码和之前章节中的批量代码，能够很好地说明这一点。

速度层中剩下的要学习的是另一种流处理的范式——微批量流处理。微批量流处理相比一次一个流处理做出了不同的取舍，比如牺牲延迟，但它有一些强大的功能，比如适用于更通用操作集合的有且仅有一次处理语义。

```

11 (flush) {
12     // batch updates to Cassandra here
13     for (Tuple t : buffer) {
14         collector.write(t);
15     }
16     buffer.clear();
17 }
18
19 public void flush(CassandraBatchWriter batcher) {
20     // flush the batcher
21     batcher.flush();
22 }

```

注意：拓扑是完全容错的。如果数据源或数据接收器发生故障，系统会自动重试，直到成功为止。

所以任何失败都将导致 Spout 元组回放。失败和重试不会影响系统的准确性，因为增量流处理中，每个元组都会被处理多次。元组被处理多次，在分布式系统中，这是一个非常常见的现象。

在分布式系统中，每个元组都会被处理多次。元组被处理多次，在分布式系统中，这是一个非常常见的现象。

在分布式系统中，每个元组都会被处理多次。元组被处理多次，在分布式系统中，这是一个非常常见的现象。

在分布式系统中，每个元组都会被处理多次。元组被处理多次，在分布式系统中，这是一个非常常见的现象。

在分布式系统中，每个元组都会被处理多次。元组被处理多次，在分布式系统中，这是一个非常常见的现象。

在分布式系统中，每个元组都会被处理多次。元组被处理多次，在分布式系统中，这是一个非常常见的现象。

在分布式系统中，每个元组都会被处理多次。元组被处理多次，在分布式系统中，这是一个非常常见的现象。

在分布式系统中，每个元组都会被处理多次。元组被处理多次，在分布式系统中，这是一个非常常见的现象。

在分布式系统中，每个元组都会被处理多次。元组被处理多次，在分布式系统中，这是一个非常常见的现象。

在分布式系统中，每个元组都会被处理多次。元组被处理多次，在分布式系统中，这是一个非常常见的现象。

微批量流处理

本章内容

- 有且仅有一次处理语义
- 微批量处理及其取舍
- 微批量流处理的管道图扩展

前面 4 章已经介绍了速度层的主要概念：实时视图、增量算法、流处理，以及如何将这些概念组合在一起。本章将重点放在一种不同的流处理方法上，该方法通过进行取舍而从中受益，比如提高精确度和获得更高的吞吐量。

如前所述，一次一个流处理是低延迟且容易理解的。但它在失败期间只能提供“至少一次”的处理保证。虽然对于特定的操作，比如添加元素到集合，这种方法并不会影响精确度，但它的确会影响其他操作，比如计数操作的准确性。在许多情况下，不准确性是不重要的，因为批处理层覆盖速度层，所以该不准确性只是暂时的。但是在一些情况下，你需要在任何时间都有完整的准确性，而且暂时的不准确性也是不可接受的。在这些情况下，使用微批量流处理可以实现所需要的容错准确性，代价是大约数百毫秒到秒的更高延迟。

在深入讨论微批量流处理的思想之后，本章会介绍如何将用于批量处理的管道图扩展到用于微批量流处理。这些扩展的管道图将被用于完成 SuperWebAnalytics.com 速度层的设计。

16.1 实现有且仅有一次语义

在一次一个流处理中，元组被彼此独立地处理。故障追踪是单个元组级别的，并且回放也发生在单个元组级别。

微批量流处理所适用的领域与一次一个流处理是不同的。一个小批量的元组被同时处理，如果在处理中发生任何失败，那么整个该批处理的元组将被回放。此外，每一个批量按照严格的次序被处理。这种方法允许使用新技术来实现处理中的有且仅有一次语义，而不是依靠一次一个处理中的固有的等幂函数。

下面来看这是如何工作的。

16.1.1 强有序处理

假设只是想实时计算所有元组的计数，而且在处理期间不管有多少次失败，你都想要计数完全准确。为了弄明白如何做到这一点，让我们从一次一个处理方法开始，看看它需要什么来实现有且仅有一次处理语义。

一次一个处理的伪代码如下所示：

```
process(tuple) {  
    counter.increment()  
}
```

这段代码没有实现有且仅有一次语义。考虑故障发生时的情形：元组将被回放，并且当到达增量计数时，你不知道该元组是否已经被处理。有一种情况很可能发生——你增加了计数然后在元组被确认之前突然发生失败。唯一能知道是否已经对元组计数的方法是存储每个已经处理的元组 ID——但这样会存储大量的状态而不只是一个数字。所以这不是一个很可行的解决方案。

实现有且仅有一次语义的关键是在输入流上执行强有序处理。下面来看当一次只在输入流上处理一个元组，并且直到当前元组被成功处理才移动到下一个元组时会发生什么。当然，这不是一种可扩展的解决方案，但它说明了微批量处理背后的核心理念。此外，我们假设每个元组都有一个与之相关联的唯一 ID，不管元组被回放多少次，ID 总是相同的。

核心理念是存储计数和最近被处理的元组 ID，而不是仅仅存储一个计数。现在，你更新计数时，有以下两种情况：

- ❑ 存储的 ID 与当前的元组 ID 相同。在这种情况下，计数已经包含了当前的元组，所以什么也不用做。
- ❑ 存储的 ID 不同于当前的元组 ID。在这种情况下，计数并没有包含当前的元组，所

以增加计数器并更新存储的 ID。由于元组被有序处理，并且计数和 ID 被自动更新，因此这能正常工作。

这种更新策略适用于所有失败场景。如果处理进程在更新计数之后失败，那么元组将被回放并且忽略第二次更新计数。如果处理进程在更新计数之前失败，那么第二次将发生更新计数。

16.1.2 微批量流处理

正如前面提到的，一次处理一个元组是非常低效的。更好的方法是将元组作为离散的批次来处理，如图 16-1 所示。这种方法被称为微批量流处理。

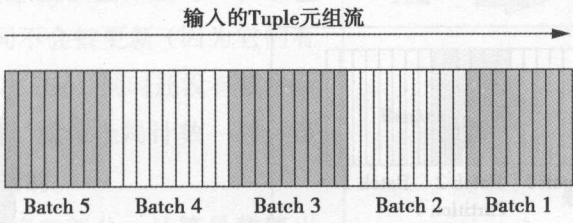


图 16-1 元组流被划分到多个批次中

每个批次被有序处理，并且每个批次都有唯一的 ID，该 ID 每次回放总是一样的。因为每次迭代有很多元组被处理而不是只处理一个，所以可以通过并行化该处理实现可扩展。在移动到下一个批次之前，该批次必须被处理完成。

下面来看使用微批量流处理的全局计数的示例是如何工作的。此外，不是只存储计数，而是存储计数以及更新计数涉及的最近批次 ID。例如，假设存储状态的当前值如图 16-2 所示。

Count	112
Batch ID	3

图 16-2 包含批次 ID 的计数状态

Count	122
Batch ID	4

图 16-3 更新计数状态的结果

现在假设数据库中的状态被更新后，流处理器中发生了一些失败，并且批处理完成的消息将不会被收到。流处理器会使该批次超时并重试批次 4。当重试批次 4 到达更新数据库中的状态时，它发现该状态已经被批次 4 更新。所以批处理器不会再一次增加计数，而是不做任何处理，并移动到下一个批次。

下面通过一个比全局计数更加复杂的示例，进一步讨论微批量流处理是如何工作的。

16.1.3 微批量流处理的拓扑结构

假设想要构建一个流应用程序，用于输入大量的单词并计算最频繁出现的前三个单词。微批量流处理可以在完全并行化、支持容错和准确性的同时完成该任务。

对于单词的每个批次需要完成两个任务：首先，必须保存每个单词的出现频率。这可以通过使用一个键/值数据库来完成。其次，如果刚处理的任何单词的频率高于当前最频繁的前三个单词，那么前三的列表必须被更新。

下面从更新频率开始。就像 MapReduce 和一次一个流处理分区数据以及并行化处理每个分区那样，微批量流处理也需要完成相同的工作。处理一个批次的单词，如图 16-4 所示，单个批次包含输入流中所有分区的元组。

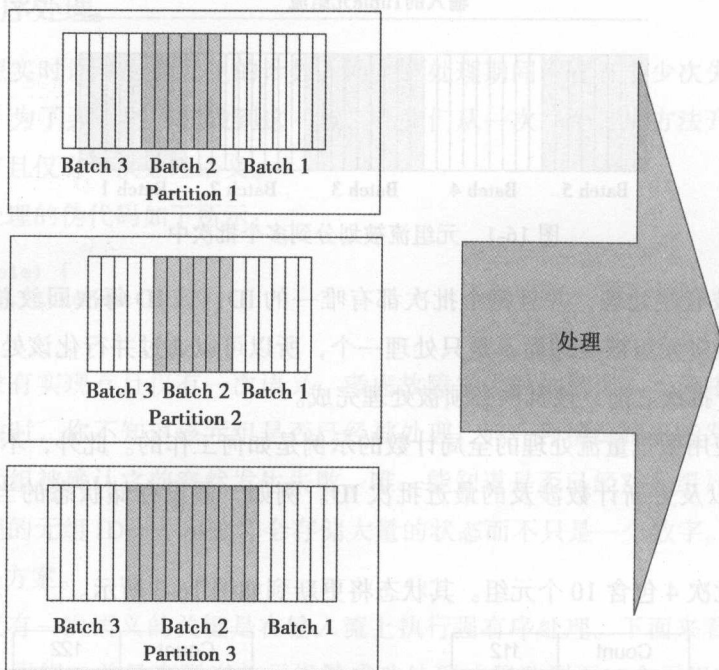


图 16-4 每个批次包括输入流中所有分区的元组

为了更新单词的频率，单词必须被重新分区，这样相同的单词才能被相同的任务所处理。这样可以确保当数据库更新时，只有一个线程将更新该单词的值，并且不会有竞态条件，如图 16-5 所示。

现在需要弄清楚存储什么作为每个单词的状态。与“全局计数只存储一个计数不够”相同，为每个单词只存储一个计数也是不够的。如果一个批次被重试，你无法知道当前计数是否包含了当前的批次。所以就像全局计数的解决方案中存储计数的批次 ID 那样，单词

计数的解决方案是存储每个单词计数的批次 ID，如图 16-6 所示。

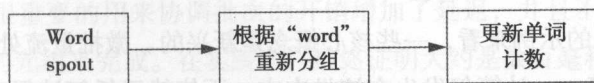


图 16-5 word-count 拓扑结构

现在考虑一个失败的场景。假设当一个批次被处理时，集群中的一台机器宕机，并且只有一些分区成功地更新了数据库。一些单词的计数被包含到当前批量的计数，而其他单词的技术还未被更新。当该批次被回放时，状态包含当前批次 ID 的单词不会被更新（因为它们有和当前批次相同的批次 ID），同时还没有被更新的单词将会正常更新。就像全局计数一样，该处理是完全准确和容错的。

现在进入计算的第二部分：计算最频繁出现的前三个单词。一种解决方案是将每个单词的新计数发送到一个单独的任务中，并且使该任务合并单词计数到出现频率前三的列表中。这种方法的问题是，它不是可扩展的。被发送到计算出现频率前三列表的单个任务的元组数量可能与整个单词的输入流几乎是大小相同的。

幸运的是，还有一种更好的、没有这个瓶颈的解决方案。不是发送每个更新的单词计数到一个单独的任务中，而是每个单词计数任务计算当前批次中出现频率最高的前三个单词，然后发送它的前三列表到负责全局前三的任务中。全局前三任务可以将所有这些列表合并到自己的列表中。现在，每个批次中传输到全局前三任务的数据量与单词计数任务的并行化成正比，而不是整个输入流。

现在考虑失败是如何影响前三大频繁单词部分的计算的。假设有一个失败导致其中一个前三的列表没有被发送到全局前三的任务中，在这种情况下，全局前三列表将不会更新，只有当该批次回放后，它才会正常更新。

假设在前三列表更新之后出现一个失败（即批次完成的消息从来没有到达）。在这种情况下，该批次将被回放，且全局前三的任务将收到相同的前三列表。这时，全局前三列表已经包含了当前批次的结果。但因为合并这些列表是一个幂等操作，将它们合并到一个已经更新的列表不会改变结果，所以之前使用在状态中包含批量 ID 的方法，在这种情况下不需要实现完全准确的处理。

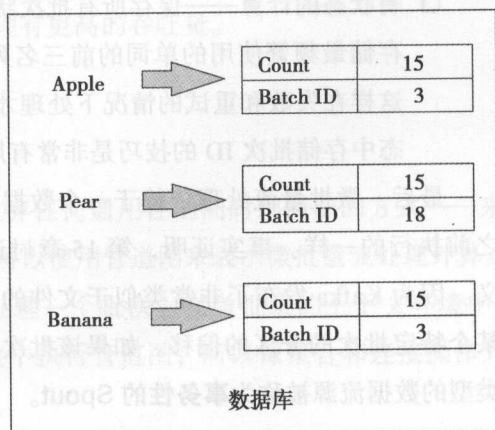


图 16-6 存储单词计数及其批次 ID

16.2 微批量流处理的核心概念

从 16.1 节所展示的示例来看，一些核心概念是新兴的。微批量流处理主要有两个方面：

- ❑ **本地批量计算**——计算仅发生在该批次中，不依赖于任何被保存的状态。这包括根据 word 字段重新对单词流分区，并计算一个批次中所有元组的计数。
- ❑ **有状态的计算**——保存所有批次状态的计算，如更新全局计数、更新单词计数，或存储最频繁使用的单词的前三名列表。对于如何完成状态更新，你必须非常谨慎，这样在失败和重试的情况下处理才是幂等的。对于在非幂等操作中添加幂等性，状态中存储批次 ID 的技巧是非常有用的。

最后，微批量流处理依赖于一个数据流源，该数据流源可以回放批次执行就像该批次之前执行的一样。事实证明，第 15 章讨论过的诸如 Kafka 的队列，对于这个有完美的语义。因为 Kafka 发布了非常类似于文件的 API，当一个批次发出时，消费者可以记住读取某个特定批次的分区的偏移。如果该批次必须回放，那么可以发出完全相同的批次。这些类型的数据流源被称为**事务性的 Spout**。

事务性的 Spout 之外

即便不使用事务性的 Spout，也有办法实现有且仅有一次处理语义。在事务性的 Spout 中，每次回放时必须产生完全相同的批次。事务性 Spout 的问题是，如果一个批次失败且该批次的一个分区变得不可用，那么由于该批次不能完全回放，处理将无法继续。

一种限制较少的 Spout，被称为**不透明的 Spout**，必须确保每个元组只在一个批次中被成功处理。它允许以下的事件序列：

- 1) 带有来自分区 1、2 和 3 的元组的批次 A 被发出。
- 2) 批次 A 处理失败。
- 3) 分区 3 变得不可用。
- 4) 批次 A 只回放分区 1 和 2 中的元组。
- 5) 在晚些时候，分区 3 变得可用。
- 6) 那些之前在失败批次中的元组在后面的批次中被成功处理。

为了使用不透明的 Spout 实现有且仅有一次语义，在完成状态更新时需要更加复杂的逻辑。只存储被更新的状态的批次 ID 是不够的，还必须追踪少量的其他信息。你可以在 Apache Storm 的文档中找到更多关于不透明的拓扑结构的信息。

与一次一个流处理相比，微批量流处理的延迟和吞吐量的特点有所不同。对于任何单

独的元组，在微批量流处理中，从添加元组到源队列直至它被完全处理的延迟时间要高得多。有一个很小但很重要的用来协调批次的开销增加了延迟，并且不是只等待一个元组完成，处理需要更多的元组来完成。在实践中，最终证明大约是数百毫秒到秒的延迟。

但微批量流处理比一次一个流处理有更高的吞吐量。一次一个流处理必须完成单个元组级别的追踪，而微批量流处理只需要完成批次级别的追踪。这意味着平均每个元组需要更少的资源，这使得微批量流处理比一次性流处理有更高的吞吐量。

16.3 微批量流处理的扩展管道图

管道图提供了一个很好的方式——不需要放弃任何通用性的同时用简洁的方式——来表达批量计算。事实证明，利用一些扩展，你也可以使用管道图来表示微批量流处理计算。

到目前为止，你知道的管道图是用于一次处理一个批次。在微批量的上下文环境中，一种解释管道图的简单方法是独立地在每个批次中执行管道图，所以像聚合和连接操作只发生在一个批处理中——在批处理之间没有计算。

当然，完全彼此独立地处理每个批次不是那么有用。它能处理本地批量计算，但你也需要能够保存批次之间的状态。这需要扩展管道图，并且在这些扩展中，为了使用有且仅有一次语义逻辑，批次 ID 也要存在。

下面通过一个示例对这些扩展进行介绍。让我们再次看看 word count 示例的流实现，只是这一次使用微批量流处理。

图 16-7 展示了完成该示例的管道图。你应该像解释那些常规的批量处理一样解释这个管道图。它处理完成包含称为 sentence 的单个字段的元组的批次。该批次处理一旦完成，它将进行下一个批次元组的处理。

正如你所看到的，除了不是 Aggregator，而是一个与 MapState 相连的 StateUpdater，这几乎与批量 word count 的管道图是一样的。MapState 代表键/值存储，可以是外部键/值数据库（如 Cassandra）或保存在处理任务内存中的状态。StateUpdater 函数

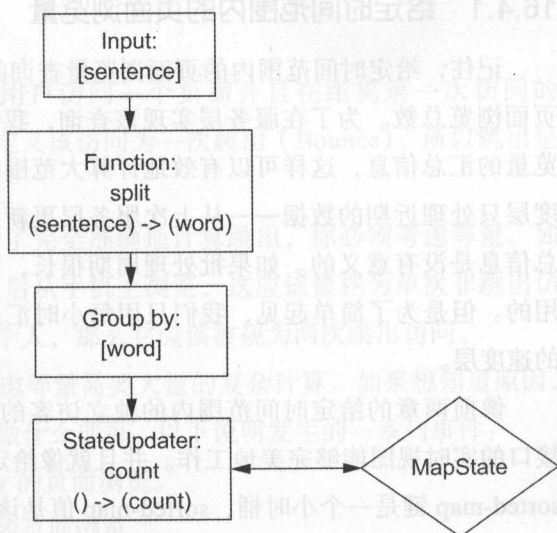


图 16-7. 微批量 word-count 管道图

负责回放时以幂等的方式来更新状态。在这种情况下，StateUpdater 函数应用 Count 聚合器来更新 MapState 中每个单词的计数。

微批量的优势是，所有促使幂等性的存储和检查批次 ID 的具体细节都可以在抽象下自动实现。这意味着在制作这些管道图时根本不需要考虑这些问题。你可以假定每个元组被处理一次。在底层，管道图以容错和回放时幂等的方式得到执行。

在管道图中，你可以存储任何类型的状态——不仅仅是 MapState。状态代表任何你需要用来满足实时索引需求的索引数据模型。例如，你可能有一个 KeyToSortedMapState，它依靠支持这种数据模型的数据库，如 Apache Cassandra。

在批量处理的上下文环境中，你知道了管道图是 MapReduce 的完整替代。管道图可以做 MapReduce 所能做的任何事情，具有相同的性能，且表达得更优雅。但是在流处理的上下文环境中，管道图是表达微批量计算的唯一方式，并且不是一次一个流处理的替代品。因为微批量流处理和一次一个流处理在延迟、吞吐量和保证消息处理语义之间做出了不同的取舍，并不存在那种“对所有情况都是最好的”方案。

16.4 完成 SuperWebAnalytics.com 的速度层

下面利用微批量流处理来完成 SuperWebAnalytics.com 速度层的设计。让我们先从给定时间范围内的页面浏览量的速度层开始。

16.4.1 给定时间范围内的页面浏览量

记住，给定时间范围内的页面浏览量查询的目标是获得任何小时范围内的一个 URL 的页面浏览总数。为了在服务层实现该查询，我们计算每小时、每天、每周和每月的页面浏览量的汇总信息，这样可以有效地计算大范围的查询。速度层中不需要这种优化，因为速度层只处理近期的数据——从上次服务层更新的时间应该只是几个小时，所以保存其他汇总信息是没有意义的。如果批处理周期很长，比如超过一天，那么每日汇总信息可能是有用的。但是为了简单起见，我们只用每小时汇总信息来实现给定时间范围内的页面浏览量的速度层。

像前两章的给定时间范围内的独立访客的速度层一样，一个实现了 key-to-sorted-map 接口的实时视图能够完美地工作。并且就像给定时间范围内的独立访客一样，key 是 URL，sorted-map 键是一个小时桶，sorted-map 值是该 URL 和该小时桶的页面浏览量。图 16-8 所示的管道图实现了该逻辑。

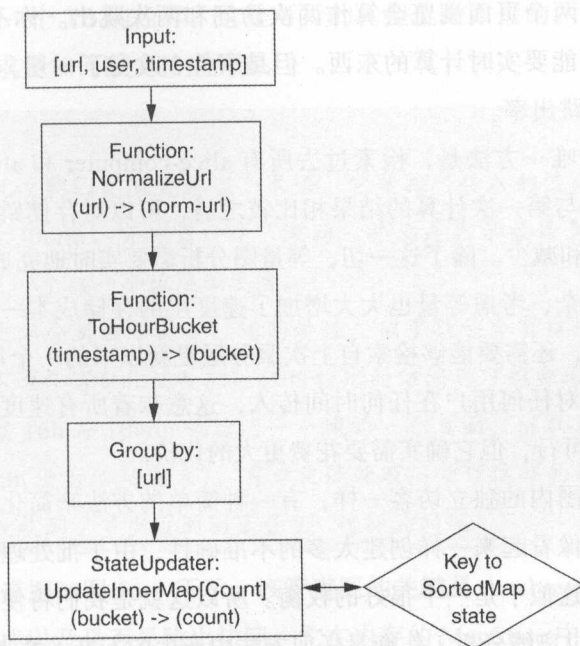


图 16-8 微批量给定时间范围内的页面浏览量的管道图

该管道图使用 key-to-sorted-map 的状态，并且在页面浏览传入时更新状态。UpdateInnerMap 状态更新程序使用 Count 聚合器进行参数化，所以它知道如何将更新应用到状态的内部映射中，该映射代表小时桶的页面浏览计数。

16.4.2 跳出率分析

现在继续在速度层计算跳出率。如果用户访问一个页面并且在距离第一次访问的 30min 内没有访问该域的另一个页面，那么定义该访问为一次跳出（Bounce）。所以跳出是基于事件没有发生而发生的。

与给定时间范围内的独立访客一样，为了完全准确地计算跳出，你必须考虑等量。如果用户在计算机上浏览了网站，然后 10min 后从手机上浏览，这应该被视为单次非跳出访问。如果你不知道这两个用户标识符是同一个人，那么它应该被视为两次跳出访问。

与给定时间范围内的独立访客一样，考虑等量需要大量的复杂计算。如果想知道原因，那么只要考虑一下当接收到新等量时你需要做什么即可。以下说明发生的一系列事件：

- 1) 在 0min 时接收到来自 alice-computer 的页面浏览。
- 2) 在 10min 时接收到来自 alice-phone 的页面浏览。
- 3) 在 140min 时接收到 alice-computer 和 alice-phone 之间的等量。

事件 3 之前, 这两个页面浏览会算作两次访问和两次跳出。你不知道它们是等价的, 所以没有什么其他可能要实时计算的东西。但是事件 3 改变了一切。现在你必须返回并修改两个小时计算的跳出率。

完成这项工作的唯一方法是, 检索过去所有 `alice-computer` 和 `alice-phone` 的访问并重新计算适当的跳出。与第一次计算的结果相比较之后, 可以对存储跳出计数和访问计数的数据库做适当的增加和减少。除了这一切, 等量图分析需要实时地完成。

除了实现更加复杂, 考虑等量也大大增加了速度层的存储成本——不仅需要存储每个域的跳出和访问数量, 还需要能够检索自上次服务层更新以来的一个用户的所有页面浏览。因为一个等量可能会对任何用户在任何时间传入, 这意味着所有速度层的页面浏览都需要被索引。这并不是不可行, 但它确实需要花费更大的代价。

就像给定时间范围内的独立访客一样, 有一种简单的方法来简化这一切: 在速度层中忽略等量。这并不会像看起来一样创建太多的不准确性, 由于批处理层和服务层会自动纠正速度层的不准确, 这似乎是一个很好的权衡。所以这就是我们将使用的方法。当然, 像往常一样, 你应该通过离线分析工作衡量任何方法中产生的不准确性来验证这些方法。

实时完成跳出率分析有 3 个步骤:

- 1) 使用包含用户标识符、URL 和时间戳的页面浏览事件流。
- 2) 通过检测用户访问一个域的一个页面并且在 30min 内没有访问其他页面来确定跳出。
- 3) 更新包含从域到跳出率映射的数据库。跳出率将存储为数字对 [跳出次数, 访问次数]。

与给定时间范围内的页面浏览量一样, 跳出率分析开始于页面浏览量流的使用。然而确定跳出更加有趣, 因为它是基于时间的事件而不仅仅是简单的聚合。在任何给定的时刻, 都可能会发生一次跳出, 这将是基于 30min 之前的页面浏览。因为在流处理中, 你只能访问刚刚发生的事件, 为了确定跳出, 你需要保存过去 30min 所发生事情的状态。

主要的思想是追踪每一次访问 (一个 [域名, 用户] 对), 直到该访问完成。正如已经定义的, 当 30min 过去而用户在该域上没有进一步的页面浏览, 那么该次访问完成。访问一旦完成, 该域的跳出率信息就可以被更新。每个完成的访问增加了一次访问量, 并且如果该次访问只有一个页面, 那么跳出数量也增加。在管道图级别, 为了追踪这些访问, 我们将保存一个映射从 [域名, 用户] 到该访问中发生页面浏览的初次时间和末次时间。一分钟一次地扫描这些访问, 我们将遍历整个映射来确定哪些访问已经完成。已经完成的访问将从映射中被移除, 然后被用来更新对应域的跳出率信息。

该策略需要追踪过去 30min 的所有访问。如果规模很大, 那么可能同时有大约数亿甚至数十亿的访问。如果追踪一次访问需要约 100B 的内存用来存储域、用户 ID、时间戳和

映射, 那么需要大约一个 TB 的内存。这是可行的, 但也是昂贵的。完成基于内存的设计后, 你将看到内存需求是如何减少甚至消除。

窗口化流处理?

之前你可能听说过窗口化流处理这个术语, 它是指将传入的流分解成时间窗口, 如 30s、1min、5min 或 30min。有时窗口是“滚动”并总是指示最后 X 秒的时间。其他时候窗口是固定的并且一个接一个地准确发生。

乍看起来, 跳出率分析由于面向时间的特性, 似乎很适合窗口化流处理, 但深入思考会发现它根本并不适合窗口化流处理。任何特定的访问都可能会跨越无限期的时间段。例如, 如果有人连续 16h 每 10min 地访问一个域的一个页面, 那么该访问必须在映射中保存 16h (直到 30min 没有活动发生)。窗口化流处理不处理这样的计算——而是用来回答“过去 15min 我收到了多少页面浏览量?”这样的问题。

跳出率分析的管道图如图 16-9 所示。该管道图的关键是 AnalyzeVisits 状态更新器, 它决定访问什么时候完成以及是否是跳出访问。它在内存的 MapState 中保存访问的状态。

下面是状态更新器的伪代码:

为每次访问存储时间戳对 [最初访问时间, 最末访问时间]。
访问信息存储在键为 [域名, 用户] 对的映射中。

```
function AnalyzeVisits(mapstate, domain, user, timestamp) {
  THIRTY-MINUTES-SECS = 60 * 30
```

```
    update(mapstate,
           [domain, user],
           function(visit-info) {
             if(visit-info == null) [timestamp, timestamp]
             else [visit-info[0], timestamp]
           })
    last-sweep-time = get(state, "last-sweep", 0)
```

获取上次检查的访问, 以确定它们是否已经完成 (自从上次访问的 30min 未产生访问时间的)。这些访问将一分钟一次地被扫描。

当一次访问完成时, 发出这个事实并在映射中停止追踪该次访问。

```
    if(timestamp > last-sweep-time + 60) {
      for(entry in mapstate) {
        domain = entry.key[0]
        visit-info = entry.value
        if(timestamp > visit-info[1] + THIRTY-MINUTES-SECS) {
          emit(domain, visit-info[0] == visit-info[1])
          remove(mapstate, entry.key)
        }
      }
      put(mapstate, "last-sweep", timestamp)
    }
  }
```

自上次检查后, 当超过 1min 时, 扫描所有访问。注意: 这段代码取决于传入页面浏览的稳定的流, 因为它只在检查到新的页面浏览时间戳之后才进行扫描。

如果最初和最末访问时间是相同的, 那么该次访问是跳出访问。

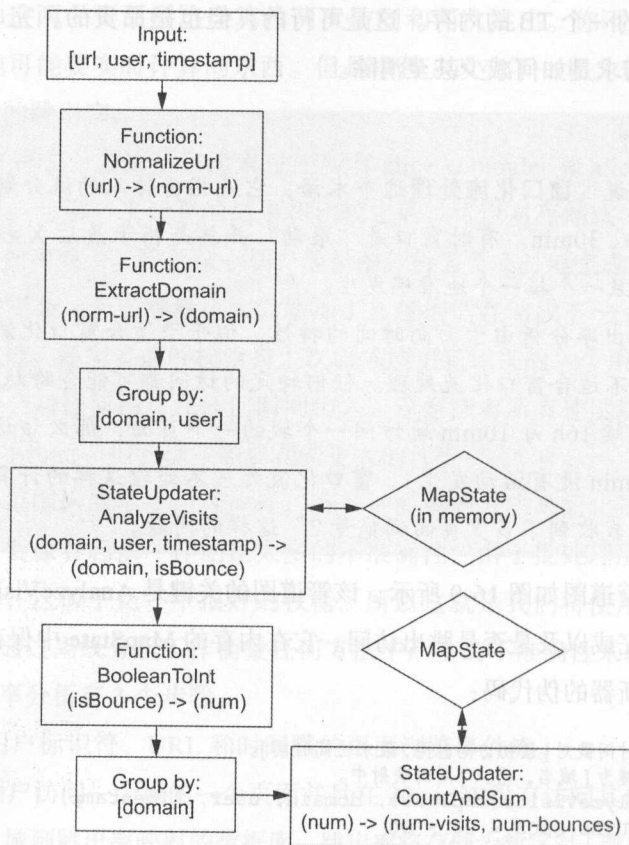


图 16-9 微批量跳出率分析的管道图

AnalyzeVisits 发出访问信息的流：一个包含域和一个布尔值标志该访问是否是跳出访问的二元组。管道图的下一部分计算访问总次数和跳出总次数，并将它们存储在 MapState 中。首先它将布尔值转换成 0 或 1，这取决于它是假或真。它对元组计数来确定访问的总次数，并且对 0 和 1 的总数进行相加以确定跳出的总次数。这两个数字存储在一起作为 MapState 的值，以键作为域。

时间和无序的消息

跳出率分析管道图中的一个假设是，元组中的时间戳总是增加的。但如果元组是无序地传入，该怎么办？

这并非是一个假想的问题——元组是在整个集群上生成，然后一起放入队列服务器的，因此很可能不是完美有序的。除此之外，如果有网络分区或错误，并且写元组到队列的任务有延迟，那么元组可能是更长时间的无序，甚至是分钟级别的。

对于诸如给定时间范围内的页面浏览量的很多计算，这并不重要。但对于跳出率分析，它采用时间来触发检查完成的访问，这是很重要的。例如，在已经检查完成的访问之后，访问的页面浏览也可以传入，并且该页面浏览可能会将完成的访问转变为仍然活跃的申请。

处理无序元组的方法是在计算中引入延迟。对于跳出率分析的代码，你可以将一次完成的访问定义改变为“自从上次访问已经过去了 45min，并在上次页面浏览之后的 30min 内没有额外的页面浏览”。该策略将处理晚传入 15min 的无序元组。

当然，对于无序的元组，没有完美的处理方法。理论上你可以接收到两天前生成的元组，但无限期等待来确定是否有任何无序元组的跳出率分析代码是不合理的，这样会使代码无法取得任何进展。你必须对愿意等多久做出限制。像往常一样，通过确定无序元组的分布和比例来限制等待时间是谨慎的。

与速度层中的许多事情一样，这是从根本上难以实时处理的其他一些示例，但在批处理层中这并不是问题。因为 Lambda 架构有覆盖速度层的批处理层，任何无序元组引入的不准确性都会随着时间被消除。

但是这样设计有一个弱点——所有状态被保存在执行管道图任务的内存中。之前我们计算出非常大的规模需要 TB 级别的内存。虽然拥有这么多内存的集群是可能的，但事实证明还有另一个不需要任何内存的方法。

16.5 另一个跳出率分析示例

下面来看如何能完全消除跳出率分析的内存需求。技巧是退后一步并再次查看这个问题。直到 30min 过去且没有关于该访问的活动，该访问才完成。你必须等待 30min 来确定访问的状态，这意味着跳出率分析从根本上来说不是一个实时的问题。延迟并不是一个很大的约束，所以根本无须使用一个侧重内存的流处理系统。

对于这个问题，为速度层使用批处理系统是绝对可行的。工作流的逻辑不需要改变——你仍然要维护从访问到关于访问的信息的映射，在非活跃的 30min 过去之后，标记访问为跳出或非跳出访问，然后将跳出和访问信息聚合到键/值数据库中。所不同的是底层技术的变化：对于计算系统，你可以使用 Hadoop；对于存储速度层视图，你可以使用像 ElephantDB 的服务层数据库。最后，对于中间键/值状态，你也可以使用类似 ElephantDB 的数据库。使用增量批处理的跳出率分析如图 16-10 所示。

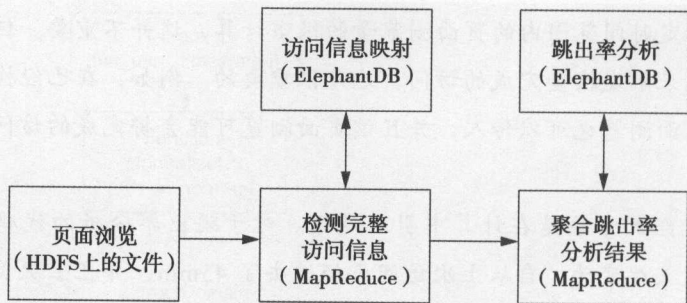


图 16-10 使用增量批处理的跳出率分析

到目前为止，我们只讨论了使用批量计算来做完整的重新计算，即在批量计算中一次性使用所有数据来从头开始生成一个视图。增量的批处理工作原理则是使用新的数据并基于上个版本的视图来产生新的视图。与流处理相比，它恰当地改变了视图。增量批处理工作流产生的视图是全新的，并且创建之后永远不会被修改。本书将在第 18 章进一步讨论增量批处理。

16.6 总结

你已经知道通过牺牲一些延迟，可以超越一次一个流处理的至少一次语义，实现有且仅有一次处理语义。一次一个流处理只对固有的幂等操作有且仅有一次语义，而微批量流处理可以对几乎任何计算实现有且仅有一次语义。

很明显，速度层并不一定意味着实时，也不一定需要流处理。速度层是关于近期数据的——你从跳出率分析的示例中看到的对问题的定义，本质上就不是实时的。由于速度层只负责近期数据的处理，因此允许使用其他方法，比如增量的批处理。

现在你已经理解了微批量流处理的概念，那么应该准备好在实践中应用这项技术了。

微批量流处理：示例

本章内容

- ❑ Trident, Apache.Storm 的微批量处理 API
- ❑ 整合 Kafka、Trident 和 Cassandra
- ❑ 容错任务的本地状态

第 16 章介绍了微批量处理的核心概念。通过将元组分成一系列较小的批次进行处理，你可以实现有且仅有一次处理的语义。通过维护批量处理的强有序性并在状态中存储批量 ID 信息，你可以知道该批量之前是否已经被处理过。这可以避免多次应用更新，从而实现有且仅有一次语义。

如前所述，一些稍经扩展的管道图可以被用来表示微批量流计算。这些管道图让你认为计算中的每个元组好像只被处理了一次，但是这些管道图可以编译代码，用来自动处理失败、重试和所有批量 ID 逻辑的具体细节。

现在，你将了解 Apache Storm 的微批量处理 API-Trident，Trident 提供了一种这些扩展管道图的实现。你将看到它与正常的批处理是多么类似。你将了解如何将它与 Kafka 之类的流消息源和 Cassandra 之类的状态提供者整合在一起。

17.1 使用 Trident

Trident 是一个 Java API，它将微批量处理的拓扑结构翻译成 Storm 的 Spout 和 Bolt。

Trident 看起来非常类似于你已经熟悉的批处理用语——它有连接、聚合、分组、函数和过滤器。此外，它增加了使用任何数据库或持久性存储完成跨批量的状态性处理的抽象。

图 17-1 所示为 word count 流的管道图。下面来看如何使用 Trident 实现该管道图。

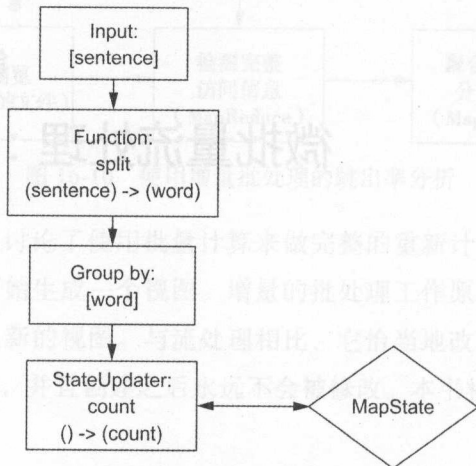


图 17-1 微批量 word-count 的管道图

出于演示目的，该示例将从下列的源中读取无限的句子流：

```

FixedBatchSpout spout = new FixedBatchSpout(
    new Fields("sentence"),
    3, // number of tuples in each batch
    new Values("the cow jumped over the moon"),
    new Values("the man went to the store"),
    new Values("four score and seven years ago"),
    new Values("how many apples can you eat"),
    new Values("to be or not to be the person"));
spout.setCycle(true); // repeats these tuples forever
  
```

该 Spout 的每个批处理发出 3 个句子并无限循环这些句子。

下面是实现 word count 的 Trident 拓扑的定义：

```

TridentTopology topology = new TridentTopology();
topology.newStream("spout1", spout)
    .each(new Fields("sentence"),
        new Split(),
        new Fields("word"))
    .groupBy(new Fields("word"))
    .persistentAggregate(
        new MemoryMapState.Factory(),
        new Count(),
        new Fields("count"));
  
```

让我们逐行浏览一下代码。首先，创建一个 TridentTopology 对象。TridentTopology 公

开了构建 Trident 计算的接口。TridentTopology 有一个名为 `newStream` 的方法，它将拓扑连接到一个输入源。在这种情况下，输入源就是之前定义的 `FixedBatchSpout`。如果你想从 Kafka 进行读取，那么要使用 `Trident Kafka Spout`。Trident 追踪 ZooKeeper 中每个输入源的少量状态（关于它所消费数据的元数据），这里的 “`spout1`” 字符串指定了 Trident 应该保持元数据的 ZooKeeper 节点。该元数据包含了每个批量中具体内容的信息，这样如果一个批量必须被重新处理，那么下次将会发出完全相同的批量。

该 Spout 发出包含一个名为 `sentence` 字段的流。拓扑定义的下一行将 `Split` 方法应用到流中的每个元组，获取 `sentence` 字段并将其分割成单词。每个句子元组可能会创建很多单词元组——比如，“`the cow jumped over the moon`” 这个句子创建了 6 个单词元组。`Split` 的定义如下：

```
public static class Split extends BaseFunction {
    public void execute(TridentTuple tuple,
                       TridentCollector collector) {
        String sentence = tuple.getString(0);
        for(String word: sentence.split(" ")) {
            collector.emit(new Values(word));
        }
    }
}
```

与 Storm Bolt 将所有元组作为输入并生成所有元组作为输出不同，Trident 操作将部分元组作为输入，并将它们的输出值附加到输入元组——这正是管道图操作设定的工作。在后台，Trident 将尽可能多的操作一起编译为单个 Bolt。

你已经看到管道图是如何通过 `StateUpdater` 操作进行扩展的，`StateUpdater` 与 `State` 对象保持通信来保存批量之间的持久化状态。Trident 有如下的 `StateUpdater` 接口：

```
public interface StateUpdater<S extends State>
    extends Operation {
    void updateState(
        S state,
        List<TridentTuple> tuples,
        TridentCollector collector);
}
```

它接受一个批量的元组，期望通过执行适当的逻辑来更新状态。

Trident 提供了两种将 `StateUpdater` 插入拓扑的方法：第一种是 `partitionPersist`，它需要获取 `StateUpdater` 接口的一个实现；第二种是 `persistentAggregate`，它需要一个 `Aggregator`。

`Aggregator` 没有状态的概念，所以 `persistentAggregate` 将把 `Aggregator` 转换成 `StateUpdater`。例如，`Count` 聚合器将被转化为添加当前批量的计数到存储在状态中的计数。这通常是非常方便的。

为了完成 word-count 的示例，拓扑的其余部分计算单词计数并保持结果的持久存储。首先，流根据 word 字段进行分组。然后，每组通过使用 Count 聚合器和 persistent-Aggregate 被持久化地聚合。在这个示例中，单词计数被保存在内存中，但是使用 Memcached、Cassandra 或任何其他持久的存储来交换单词计数是很简单的。

让我们看看如何让这段代码转而把单词计数存储在 Cassandra 中。代码如下：

```
CassandraState.Options opts =
    new CassandraState.Options();
opts.globalCol = "COUNT";
opts.keySerializer = StringSerializer.get();
opts.colSerializer = StringSerializer.get();
stream.groupBy(new Fields("word"))
    .persistentAggregate(
        CassandraState.transactional(
            "127.0.0.1",
            "mykeyspace",
            "mycolumnfamily"),
        new Count(),
        new Fields("count"));
```

该 CassandraState 实现允许使用一元组的组或二元组的组来完成分组的聚合。一元组的情况下将 Cassandra 看作键/值数据库，而二元组的情况下将 Cassandra 看作键到映射的数据库。在一元组的情况下，如前面的示例所示，元组的值对应于 Cassandra 的键，并且使用的列是选项中指定的 globalCol；在二元组的情况下，分组元组的第一个元素是 Cassandra 的键，第二个元素是 Cassandra 的列。

关于 CassandraState 的更多信息

本书附带的源代码提供了 CassandraState 的一个简单实现。然而它是不理想的，因为它一次只完成一个数据库操作而不是批量的操作，所以 CassandraState 的可能吞吐量远低于它可以的吞吐量。但是这种方式的代码更容易遵循，所以我们希望它可以作为一个参考实现，得到与你选择使用的任何数据库进行交互的状态。

下面是 Count 聚合器的定义：

```
public static class Count
    implements CombinerAggregator<Long> {
    public Long init(TridentTuple tuple) {
        return 1L;
    }

    public Long combine(Long val1, Long val2) {
        return val1 + val2;
    }
}
```

```

public Long zero() {
    return 0L;
}

```

正如你所看到的，这是 Count 的一个简单实现，类似于 JCascalog 中定义的并行聚合器。需要特别注意所有这些代码中实现有且仅有一次语义的棘手的批量 ID 逻辑。Trident 在后台自动实现这些处理。在这种情况下，它会自动存储批量 ID 与计数，如果它检测到存储的批量 ID 与当前批量 ID 相同，就不会对持久性存储做任何更新。

关于 Trident 中尚未讨论的部分，我们将在继续学习的过程中进行解释。关于 Trident API 的更深层次的信息，请参考 Storm 的在线文档。本书只展示如何将微批量流处理应用于实际问题，而不是探究这些 API 是如何工作的每一个细节中。

17.2 完成 SuperWebAnalytics.com 的速度层

下面将前一章的管道图转化为使用 Trident 的工作代码。余下两个要完成的查询是给定时间范围内的页面浏览量和跳出率分析。

17.2.1 给定时间范围内的页面浏览量

给定时间范围内的页面浏览量的管道图如图 17-2 所示。为了实现该管道图，你必须决定为源数据流和状态使用什么样的特定技术。

源数据流可由 Apache Kafka 很好地处理。记住，在失败期间实现有且仅有一次语义的关键之一是，重新执行的批量总应与之前处理过的相同。Storm 把源队列的该属性当作事务性语义。Kafka 具备这样的能力，这使它成为微批量处理的一个不错选择。

对于状态，它需要键到排序映射的索引类型。这正是 Apache Cassandra 提供的索引类型，这使之成为该应用程序的一个不错的选择。

为了实现该拓扑，第一步是定义从 Apache

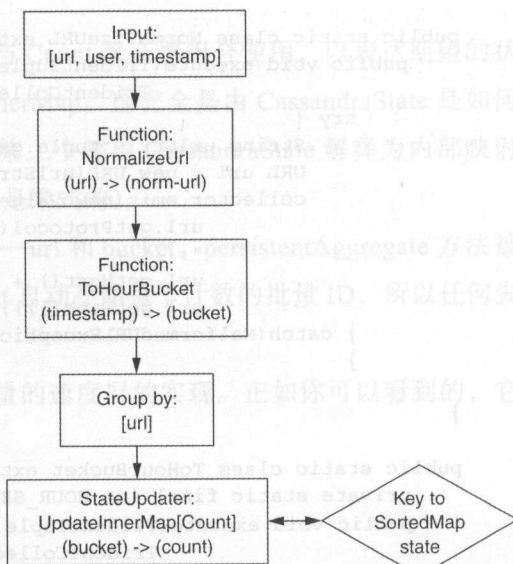


图 17-2 微批量给定时间范围内的页面浏览量的管道图

Kafka 读取页面浏览的 Spout。实现代码如下：

```
TridentTopology topology = new TridentTopology();
TridentKafkaConfig kafkaConfig =
    new TridentKafkaConfig(
        new KafkaConfig.ZkHosts(
            "zkstr", "/kafka"),
        "pageviews"
    );
kafkaConfig.scheme = new PageviewScheme();
```

配置 Trident Kafka Spout 类似于配置普通的 Storm Kafka Spout，如第 16 章所示。注意：该模式的设置将反序列化页面浏览为 3 个字段：url、user 和 timestamp。

下面是拓扑的第一部分，它规范化 URL 并将时间戳转换成适当的小时桶：

```
Stream stream =
    topology.newStream(
        "pageviewsOverTime",
        new TransactionalTridentKafkaSpout(
            kafkaConfig))
        .each(new Fields("url"),
            new NormalizeURL(),
            new Fields("normurl"))
        .each(new Fields("timestamp"),
            new ToHourBucket(),
            new Fields("bucket"))
```

正如你看到的，它只是每个任务的一个函数。下面是这些函数的实现：

```
public static class NormalizeURL extends BaseFunction {
    public void execute(TridentTuple tuple,
        TridentCollector collector) {
        try {
            String urlStr = tuple.getString(0);
            URL url = new URL(urlStr);
            collector.emit(new Values(
                url.getProtocol() +
                "://" +
                url.getHost() +
                url.getPath()));
        } catch (MalformedURLException e) {
        }
    }
}

public static class ToHourBucket extends BaseFunction {
    private static final int HOUR_SECS = 60 * 60;
    public void execute(TridentTuple tuple,
        TridentCollector collector) {
        int secs = tuple.getInteger(0);
        int hourBucket = secs / HOUR_SECS;
        collector.emit(new Values(hourBucket));
    }
}
```


该逻辑与批处理层使用的没有什么不同，在层之间共享代码是很优越的（这里的重复只是因为它更容易遵循）。

最后，剩下的是将页面浏览计数汇总到 Cassandra 中，并确保它以幂等的方式来完成。首先，让我们配置 `CassandraState`：

```
CassandraState.Options opts =
    new CassandraState.Options();
opts.keySerializer = StringSerializer.get();
opts.colSerializer = IntegerSerializer.get();

StateFactory state =
    CassandraState.transactional(
        "127.0.0.1",
        "superwebanalytics",
        "pageviewsOverTime",
        opts);
```

为键（URL）和列（时间桶）设置适当的序列化器，然后配置状态来指向适当的集群、键空间和列族。

下面是该拓扑剩余部分的定义：

```
stream.groupBy(new Fields("normurl", "bucket"))
    .persistentAggregate(
        state,
        new Count(),
        new Fields("count"));
```

在管道图中，`UpdateInnerMap` 状态更新器与 `Count` 聚合器组合使用，以表达期望的状态转换。不过在这段代码中没有提及 `UpdateInnerMap`。这完全是由 `CassandraState` 是如何工作决定的。当使用两个键完成一个分组时，第二个键由 `CassandraState` 解释为内部映射键，这意味着 `UpdaterInnerMap` 在此拓扑定义中是隐式的。

在这种情况下，分组的键包含两个字段——`url` 和 `bucket`。`persistentAggregate` 方法被用来应用内置的 `Count` 聚合器积累计数。`Trident` 自动存储每个计数的批量 ID，所以任何失败和重试都能够以幂等的方式完成。

这样就完成了给定时间范围内的页面浏览量的速度层的实现。正如你可以看到的，它非常简洁明了。

17.2.2 跳出率分析

下面来看如何使用 `Trident` 实现跳出率分析。这里以图 17-3 所示的管道图作为参考。

这是一个更复杂的拓扑结构，所以让我们一点一点地讨论它。该拓扑几乎完全反映了这个管道图。

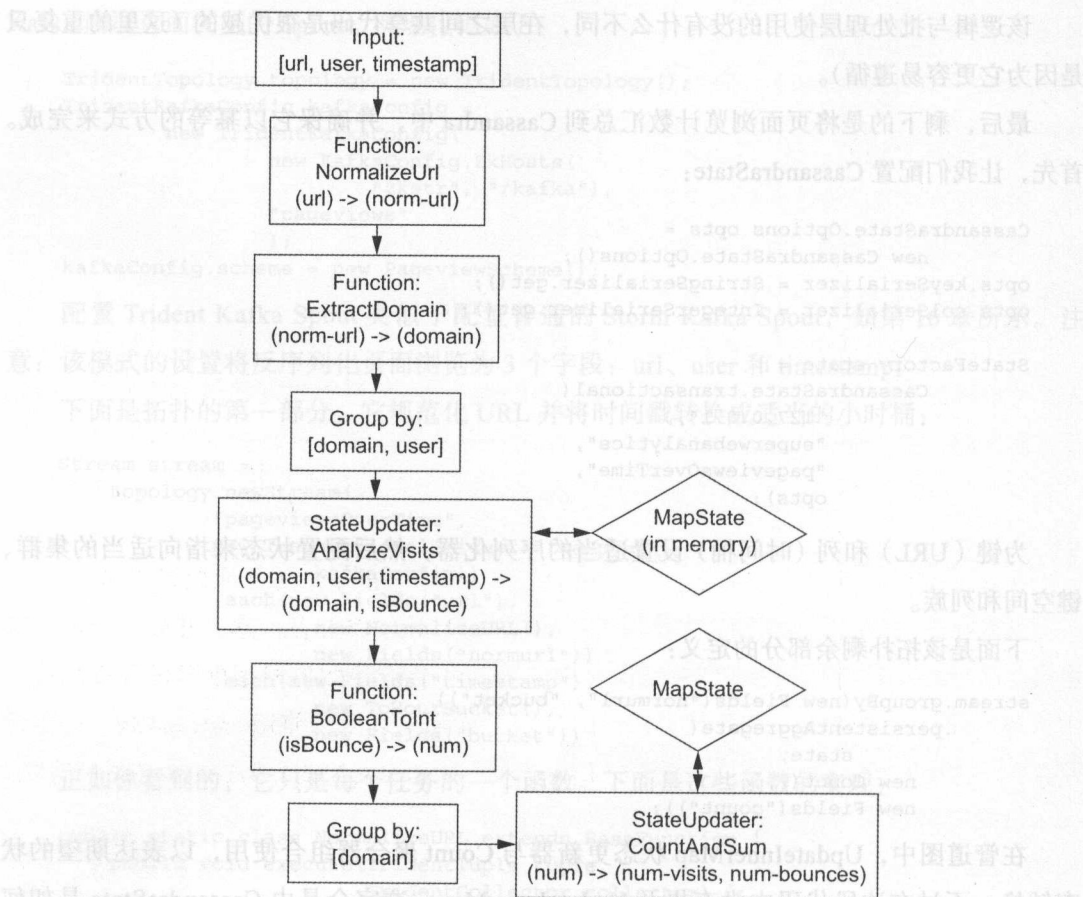


图 17-3 微批量跳出率分析的管道图

该拓扑的开始部分如下所示：

```

topology.newStream(
    "bounceRate",
    new TransactionalTridentKafkaSpout(kafkaConfig))
    .each(new Fields("url"),
        new NormalizeURL(),
        new Fields("normurl"))
    .each(new Fields("normurl"),
        new ExtractDomain(),
        new Fields("domain"))

```

这里没有什么新的内容。它消费来自 Kafka 的页面浏览流，并通过运行两个方法从 URL 中提取域。

下面是拓扑的下一部分，它分析访问并决定跳出何时发生：

```

.partitionBy(new Fields("domain", "user"))
.partitionPersist(
    new MemoryMapState.Factory(),
    new Fields("domain", "user", "timestamp"),
    new AnalyzeVisits(),
    new Fields("domain", "isBounce"))
.newValuesStream()

```

你应该注意这段代码中的一个新操作：`partitionBy`。为了弄明白为什么需要该操作，我们先来回顾 `AnalyzeVisits` 是如何工作的。`AnalyzeVisits` 一次查看一个页面浏览事件，并更新关于用户已经访问该域多长时间的状态。`AnalyzeVisits` 也一分钟一次地扫描所有访问——使用页面浏览事件中的时间戳来确定什么时候已经过去一分钟——来确定访问是否已经完成（超过 30min 该用户没有页面浏览）。因此，尽管 `AnalyzeVisits` 基于一个特定的域和一个特定的用户来更新状态，在处理单个元组时它可能会查看其状态中的所有域/用户对。

现在让我们回到 `partitionBy` 这个操作。`Trident` 提供了两种划分元组的方式：`partitionBy` 和 `groupBy`。`groupBy` 允许使用一个公共的键将元组集合在一起，并独立于所有其他组在这些组上运行聚合。而 `partitionBy` 只允许你指定应该如何根据处理任务来划分元组。拥有相同分区键的元组将进入相同的任务中。这里之所以使用 `partitionBy`，是因为 `AnalyzeVisits` 不独立处理域/用户对，而是一分钟一次地查看当前存储在内存中的所有域/用户对。

如果只根据 `domain` 字段分区，这种拓扑也是正确的。但如果只有几个域支配数据集中的访问，那么可能会导致分布失衡。如果根据 `user` 和 `domain` 分区，那么分布几乎肯定是均匀的，因为仅凭一个用户支配数据集中的页面浏览是极其不可能的。

现在来看 `AnalyzeVisits` 的实现。它通过 `MemoryMapState` 类将其所有状态保存在一个内存映射中。`MemoryMapState` 由 `Trident` 所提供，在重试的情况下，它可以让所有批量 ID 逻辑是幂等的。所以，如果有失败并且批量被重新处理，那么 `MemoryMapState` 实现确保了更新不会被应用超过一次。`AnalyzeVisits` 代码对该实现不需要任何担心。

在了解 `AnalyzeVisits` 之前，这里还需要几个帮助类。这些表示保存在状态中的键和值被 `AnalyzeVisits` 用于追踪用户访问：

```

static class Visit extends ArrayList {
    public Visit(String domain, PersonID user) {
        super();
        add(domain);
        add(user);
    }
}

```

```

static class VisitInfo {
    public int startTimestamp;
    public Integer lastVisitTimestamp;

    public VisitInfo(int startTimestamp) {
        this.startTimestamp = startTimestamp;
        this.lastVisitTimestamp = startTimestamp;
    }

```

```

    public VisitInfo clone() {
        VisitInfo ret = new VisitInfo(this.startTimestamp);
        ret.lastVisitTimestamp = this.lastVisitTimestamp;
        return ret;
    }

```

AnalyzeVisits 的实现代码如下：

```

public static class AnalyzeVisits
    extends BaseStateUpdater<MemoryMapState> {

    static final String LAST_SWEEP_TIMESTAMP = "lastSweepTs";
    static final int THIRTY_MINUTES_SECS = 30 * 60;

    public void updateState(
        MemoryMapState state,
        List<TridentTuple> tuples,
        TridentCollector collector) {
        for(TridentTuple t: tuples) {
            final String domain = t.getString(0);
            final PersonID user = (PersonID) t.get(1);
            final int timestampSecs = t.getInteger(2);
            Visit v = new Visit(domain, user);
            update(state, v, new ValueUpdater<VisitInfo>() {
                public VisitInfo update(VisitInfo v) {
                    if(v==null) {
                        return new VisitInfo(timestampSecs);
                    } else {
                        VisitInfo ret = new VisitInfo(
                            v.startTimestamp);
                        ret.lastVisitTimestamp = timestampSecs;
                        return ret;
                    }
                }
            });

            Integer lastSweep =
                (Integer) get(state, LAST_SWEEP_TIMESTAMP);
            if(lastSweep==null) lastSweep = 0;

            List<Visit> expired = new ArrayList();
            if(timestampSecs > lastSweep + 60) {
                Iterator<List<Object>> it = state.getTuples();
                while(it.hasNext()) {
                    List<Object> tuple = it.next();
                    Visit visit = (Visit) tuple.get(0);
                    VisitInfo info = (VisitInfo) tuple.get(1);

```



```

        if(info.lastVisitTimestamp >
            timestampSecs + THIRTY_MINUTES_SECS) {
            expired.add(visit);
            if(info.startTimestamp ==
                info.lastVisitTimestamp) {
                collector.emit(new Values(domain, true));
            } else {
                collector.emit(new Values(domain, false));
            }
        }
    }
    put(state, LAST_SWEEP_TIMESTAMP, timestampSecs);
}

for(Visit visit: expired) {
    remove(state, visit);
}
}
}

```

该实现中的逻辑与第 16 章的伪代码是相同的。唯一的区别是需要 Java 语法来表达它。这段代码使用一些辅助函数与 MemoryMapState 进行交互，所以出于完整性考虑，此处给出这些辅助函数的代码：

```

private static Object update(MapState s,
    Object key,
    ValueUpdater updater) {
    List keys = new ArrayList();
    List updaters = new ArrayList();
    keys.add(new Values(key));
    updaters.add(updater);
    return s.multiUpdate(keys, updaters).get(0);
}

private static Object get(MapState s, Object key) {
    List keys = new ArrayList();
    keys.add(new Values(key));
    return s.multiGet(keys).get(0);
}

private static void put(MapState s, Object key, Object val) {
    List keys = new ArrayList();
    keys.add(new Values(key));
    List vals = new ArrayList();
    vals.add(val);
    s.multiPut(keys, vals);
}

private static void remove(MemoryMapState s, Object key) {
    List keys = new ArrayList();
    keys.add(new Values(key));
    s.multiRemove(keys);
}

```

拓扑定义剩余部分的代码如下：

```
.each(new Fields("isBounce"),
    new BooleanToInt(),
    new Fields("bint"))
.groupBy(new Fields("domain"))
.persistentAggregate(
    CassandraState.transactional(
        "127.0.0.1",
        "superwebanalytics",
        "bounceRate",
        opts),
    new Fields("bint"),
    new CombinedCombinerAggregator(
        new Count(),
        new Sum()),
    new Fields("count-sum"));
```

拓扑的该部分只是消耗 ["domain", "isBounce"] 流，并将它聚集到 Cassandra 中，以确定每个域访问量和跳出量。先使用 BooleanToInt 函数，如果 isBounce 是假的则转换为 0，如果是真的则转换为 1；然后，完成一个标准的 persistentAggregate 来更新 Cassandra。

实际上，你需要做两个聚合：一个确定访问量的计数；另一个确定跳出量的 isBounce 整数的和。因此，使用 CombinedCombinerAggregator 实用程序将 Count 和 Sum 聚合器合并成一个聚合器。该实用程序的定义如下：

```
public static class CombinedCombinerAggregator
    implements CombinerAggregator {

    CombinerAggregator[] _aggs;

    public CombinedCombinerAggregator(
        CombinerAggregator... aggs) {
        _aggs = aggs;
    }

    public Object init(TridentTuple tuple) {
        List<Object> ret = new ArrayList();
        for(CombinerAggregator agg: _aggs) {
            ret.add(agg.init(tuple));
        }
        return ret;
    }

    public Object combine(Object o1, Object o2) {
        List l1 = (List) o1;
        List l2 = (List) o2;
        List<Object> ret = new ArrayList();
        for(int i=0; i<_aggs.length; i++) {
            ret.add(
                _aggs[i].combine(
```

```

        11.get(i),
        12.get(i));
    }
    return ret;
}

public Object zero() {
    List<Object> ret = new ArrayList();
    for(CombinerAggregator agg: _aggs) {
        ret.add(agg.zero());
    }
    return ret;
}
}

```

至此，跳出率分析速度层的实现就完成了。

不过这个实现中有一个问题。虽然 Trident 和 MemoryMapState 确保了更新不会被应用超过一次，但是状态没有持久化或复制到任何地方。所以如果一个携带状态的任务死亡，那么该状态就丢失了。

处理该问题的一种方法是忽略它，接受它引入的少量不准确性，并依靠批处理层来纠正所发生的不准确性。或者，可以使用容错的内存状态来完成流处理。

17.3 完全容错、基于内存及微批量处理

当工作进程死亡时，你可以通过两种方法来恢复本地内存状态的恢复。

第一种方法是利用标准的数据库技术来保存复制存储中的提交日志。适用于该方法的技术是 HDFS 文件附加或 Kafka。当更新状态时，你可以同时将更新的内容写入日志。提交日志的流程如图 17-4 所示。

当一个任务启动时，它回放提交日志来重建内部状态。当然，提交日志是无限增长的，所以基于日志的重建状态将变得越来越昂贵。你可以通过定期压缩日志来解决这个问题。压缩是保存整个状态本身的过程，然后删除该状态结构中所有涉及的提交日志元素。存储完整状态的一项伟大技术是分布式文件系统。如果采用一分钟一次

Batch ID: 1 Operation: put Args: "somekey", "someval"
Batch ID: 1 Operation: put Args: "somekey2", "someval2"
Batch ID: 2 Operation: remove Args: "somekey2"
Batch ID: 2 Operation: put Args: "somekey3", 24
Batch ID: 2 Operation: put Args: "somekey4", "someval4"
Batch ID: 3 Operation: remove Args: "somekey4"

图 17-4 提交日志

地或者当提交日志增长到一定大小之后进行压缩的方法，那么这样的压缩策略会很简单。

还有一项实现内存中状态持久化的技术根本不涉及提交日志。回想一下，Trident 的工作方式是它以强有序性来处理批量，并且通过在状态中保存批量 ID，它可以检测该批量之间是否已经被处理过并实现有且仅有一次语义。但如果计算系统可以重试的不仅仅是前一次的批量，那会怎样——比如几分钟前的批量？这可以让你产生一些新想法。

其中一种想法是通过将内存的状态写出到其他地方（如分布式文件系统），来周期性地检查任何保存在内存中的状态。你可以一分钟一次地设置检查点。该检查点也存储了该检查点表示多少源数据流的程度。

现在，假设失败已发生 45s，并且其中一个托管状态的一个分区的任务死亡。此时，失败的任务只有最后 45s 到当前批量的状态（上一个检查点的时间）。所有其他任务都完全是当前的，因为它们没有失败。

你可以通过将源数据流倒回至 45s 之前并回放它来进行恢复。尽管该操作通常是非常昂贵的，但它也是高效的，因为只有一个分区需要恢复。所以在重新计算时，你可以跳过已经有最新状态的分区。

像提交日志的方法，这种策略需要将状态定期地完整写出。然而，它不需要将提交日志写出，这使之成为一种更好的方法。

该策略需要扩展目前 Storm 和 Trident 不能实现的微批量流处理模型。然而另一个称为 Spark Streaming 的系统能实现这种方法。

Spark 和 Spark Streaming

Spark 在谈到作为 MapReduce 替代的批处理时被提及，它可以更合理地使用内存。Spark 有另一种称为 Spark Streaming 的操作模式，它通过周期性地检查内部状态实现了微批量流处理的方法。Trident 专注于整合外部数据库，而 Spark Streaming 专注于计算状态以保存在内存中。

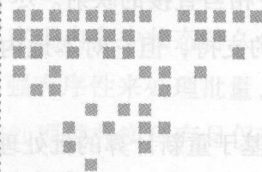
对计算系统进行分类的一个好方法是根据它们支持的计算方式来分类。三种主要的计算方式是批处理、低延迟的一次一个处理和微批量处理。Hadoop 只支持批处理，Storm 支持一次一个处理和微批量处理，Spark 支持批处理和微批量处理。

17.4 总结

本章介绍了如何使用 Storm 的 Trident API 来实际实现微批量流处理。关于数据流概念

的思考方式——管道图——和实现它的代码之间有一个相当直接的映射。尽管 Trident 对使用有且仅有一次语义在外部数据库中存储状态有出色的支持，但它对本地内存状态的支持不是完全容错的。

现在你已经探索了基本 Lambda 架构的每个层次：基于重新计算的批处理层、服务层和速度层。通过 SuperWebAnalytics.com，你已经知道了所有这些层的复杂和实现所有这些层的完整示例。有了这些基础知识之后，让我们看看如何超越基本 Lambda 架构。有许多重要的变化使你能够获得更高效的数据系统。



Chapter 18

第 18 章

深入 Lambda 架构

本章内容

- ❑ 重新讨论 Lambda 架构
- ❑ 增量批处理
- ❑ 有效管理批处理工作流中的资源
- ❑ 合并批处理和实时视图之间的逻辑

第 1 章中介绍了 Lambda 架构及其实现任意数据系统通用的方法。此后的每一章深入讨论了 Lambda 架构各种组件的细节。如你所见，构建大数据系统涉及很多东西，不仅包含扩展，也包含鲁棒性，同时也要易于理解。

现在你有机会钻研 Lambda 架构的所有不同层次，让我们使用新知识再次回顾 Lambda 架构并更好地理解它。我们将查漏补缺，并探究之前已讨论的方法的变化。

18.1 定义数据系统

我们从一个简单的问题开始：“数据系统是做什么的？”答案也很简单，“数据系统基于你所看到的过去的数据来回答问题”。或者更正式一些，“数据系统计算你已知的所有数据的函数的查询”。这对于封装任何你想构建的数据系统显然是一个直观的定义：

```
query = function(all data)
```

关于查询，应关注的属性如下：

- ❑ **延迟**——运行一个查询的时间。在许多情况下，延迟需求将会是非常低的——毫秒级别。在其他时候，一个查询花费几秒钟也是可以的。当做即席分析时，延迟需求往往是非常宽松的，甚至是小时级别。
- ❑ **时效性**——最新的查询结果如何。一个完全及时的查询需要考虑过去所见的所有数据，而不及时的查询可能不包括最近几分钟或几个小时的结果。
- ❑ **准确性**——在许多情况下，为了使查询更具有较好的性能或可扩展，你必须在查询的实现中采用近似值。

构建数据系统的很大一部分工作是使它们容错。你必须计划当遇到机器故障时，系统将如何表现。通常这意味着与前面的属性做权衡。例如，延迟和及时性之间存在基本的紧张关系。CAP 定理表明在分区的情况下，一个系统要么是一致的（查询考虑到所有以前写入的数据），要么是可用的（目前查询可以被回应）。一致性只是及时性的一种形式，可用性只意味查询的延迟是有界的。最终一致性的系统选择延迟而不是及时性（查询总是被回应，但可能不会考虑所有先前失败情况下的数据）。

因为数据系统是动态的，人们不断改变所构建的系统，并不断部署新的特性和分析，所以人是任何数据系统不可或缺的一部分。就像机器一样，人也会失败。人会将错误部署到生产中，并犯各种各样的错误。所以对于数据系统来说，允许人为错误也是很重要的。

你知道了易变性——及其相关的概念（如 CRUD）——从根本上是不容忍人为错误的。如果一个人能改变数据，那么一个错误也可以改变数据。所以允许更新和删除核心数据势必会导致数据损坏。

唯一的解决办法就是让核心数据保持不变，只允许通过写操作将新数据添加到不断增长的数据集中。你可以做一些事情，比如设置核心数据不允许被删除和更新的权限——这种冗余保证了错误不能损坏现有的数据，所以系统的鲁棒性会更好。

这把我们引向了数据系统的基本模型：

- ❑ 包含不断增长的数据集合的主数据集
- ❑ 作为函数的查询将整个主数据集作为输入

任何你想利用数据做的事都能以这种方式清晰地完成，并且这样的系统有容忍人为错误的核心重要属性。如果它是可能实现的，这将是理想的数据系统。Lambda 架构出现后，它尽可能以最小的代价实现作为对不断增长的不可变数据集的函数的理想查询。

18.2 批处理层和服务层

计算所有数据的函数的查询是不实际的，因为想在 TB 级别数据集上做的查询在几毫秒内返回是不合理的，更别说 PB 级别的数据集了。即使这是可能的，查询将是不合理的资源密集型的。对于这样一个架构，你可以做出的最简单的修改是，查询预先计算的视图而不是直接查询主数据集。这些预先计算的视图可以为查询定制，以便查询能够尽快被回应，而视图本身就是主数据集的函数。

在第 2~9 章中，你已经知道了实现这样一个系统的细节。核心是批处理系统，它能够以可扩展和容错的方式来计算所有数据上的这些函数——因此，Lambda 架构的该部分被称为批处理层。

批处理层的目的是生成索引的视图，以便这些视图上的查询能够以较低的延迟被处理。这些视图的索引和服务在服务层中完成，它与批处理层有紧密的联系。在设计批处理层和服务层时，你必须在批处理层中完成的预先计算量与视图的大小和查询时所需的计算量之间进行取舍（在第 6 章中集中讨论过）。

现在让我们越过 Lambda 架构的批处理层和服务层的基本模型，来探索设计它们时你可以有的更多选择。这些层的一个重要性能指标是需要多长时间来更新视图。因为速度层必须弥补所有没有表示在服务层中的数据，所以批处理层运行的时间越长，速度层的视图就必须越大。对具有更复杂的数据库的更大集群的需求极大地增加了操作的复杂性。此外，批处理层运行的时间越长，恢复不小心部署到生产环境中的错误的时间就越长。一种降低批处理层延迟的方法是实现增量处理。

18.2.1 增量的批处理

在第 6 章中，我们就增量算法和重新计算算法之间的权衡进行了讨论。批处理层的主要优势之一是能够利用重新计算算法，所以你可能会对使用增量算法处理批处理层的建议感到惊讶。像所有设计问题一样，为了得到最好的设计，你必须考虑所有利弊。

让我们考虑一个极端的例子——生成的唯一视图是对主数据集中所有记录的全局计数。在这种情况下，增量处理批处理层是明显胜出的，因为增量的视图不会比基于重新计算的视图更大（在这两种情况下，视图中只是一个数字），并且增加代码也并不复杂。通过没有在整个主数据集上重复地进行重新计算，你节省了大量的资源。例如，如果主数据集包含 100TB 的数据，每一个数据的新批量包含 100GB，那么批处理层的效率将会高出数个数量级。每次迭代只需要处理 100GB 的数据而不是 100TB 的数据。

现在让我们考虑另一个例子——“生日推理”的问题。在这个例子中，增量和重新计算算法之间的选择更加困难。假设你正在编写一个网络爬虫，它从人们的公开资料中收集他们的年龄。该资料不包含生日，但此时你爬取的网页中只有这个人的年龄。基于这种 [age, timestamp] 对的原始数据，你的目标是推断出每个人的生日。

生日推理算法的思想如图 18-1 所示。想象一下，你在 2012 年 1 月 4 日爬取了 Tom 的资料，并看到他的年龄是 23 岁。然后 2012 年 1 月 11 日再次爬取他的资料，并看到他的年龄是 24 岁，那么就可以推断出他的生日在这两个日期之间的某天。同样的，如果你在 2013 年 10 月 20 日爬取了 Jill 的资料并看到她的年龄是 43 岁，然后再次在 2013 年 11 月 4 日爬取她的资料看到她仍然是 43 岁，那么你知道她的生日不在这两个日期之间。你有越多年龄样品，那么就可以越好地在一个小范围内的日期内推断出别人的生日。

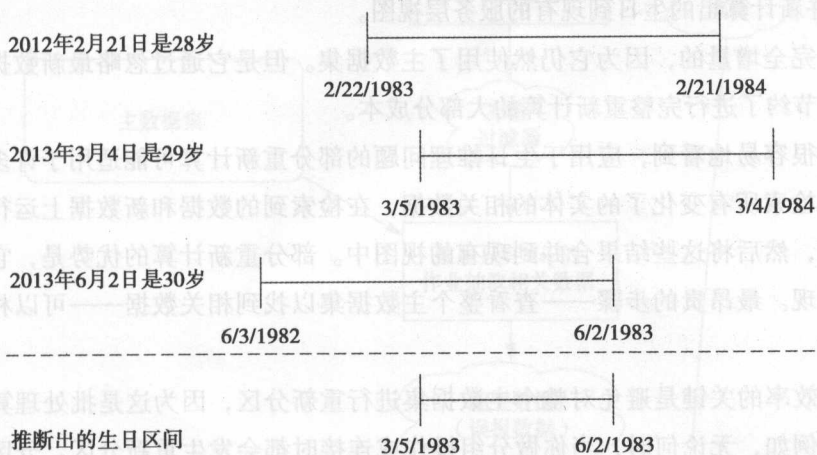


图 18-1 基本的生日推理算法

当然在现实世界中，数据可能是混乱的。有些人可能错误地输入了他们的生日，然后在稍后的日期内修改了它。这可能导致按照年龄推断算法不能得到生日，因为该年的每一天都已经消除了作为生日的可能。你可以修改生日推理算法，来搜索它可以忽略的最少数量的年龄样本，产生可能的生日的最小范围。该算法可能更愿意使用近期的年龄样本而不是更陈旧的年龄样本。

如果使用重新计算算法实现生日推理的批处理层，这是很容易的。算法可以一次性地查看一个人的所有年龄样本，做所有必要的事情来处理混乱的数据并发出一个日期的范围作为输出。但使用增量处理生日推理的批处理层是很棘手的。对于处理混乱的数据问题，不能访问全部范围的年龄样本是很难处理的。完全增量实现该算法将是相当困难的，可能需要更大、更复杂的视图。

有另一种模糊了增量和重新计算之间界线的方法，让你能够两全其美。这种技术被称为部分重新计算。

部分重新计算

每一次批处理层运行时，重新计算年龄样本中每个人的生日是很浪费的。特别是，如果自从上一次批处理层运行以来，一个人没有新的年龄样本，那么这个人的推断生日就不会改变。基于部分重新计算的生日推理批处理层的背后想法如下：

- 1) 对于新批量的数据，找出有新年龄样本的人。
- 2) 从主数据集中检索出步骤 1 中所有人的所有年龄样本。
- 3) 使用步骤 2 中的年龄样本和新批量中的年龄样本，重新计算步骤 1 中所有人的生日。
- 4) 合并新计算出的生日到现有的服务层视图。

这不是完全增量的，因为它仍然使用了主数据集。但是它通过忽略最新数据集中没有改变的人，节约了进行完整重新计算的大部分成本。

你可以很容易地看到，应用于生日推理问题的部分重新计算可能适用于许多问题。关键的想法是检索所有变化了的实体的相关数据，在检索到的数据和新数据上运行正常的重新计算算法，然后将这些结果合并到现有的视图中。部分重新计算的优势是，它们可以非常高效地实现。最昂贵的步骤——查看整个主数据集以找到相关数据——可以相对廉价地完成。

使其有效率的关键是避免对整个主数据集进行重新分区，因为这是批处理算法最昂贵的一部分。例如，无论何时，当你做分组操作或连接时都会发生重新分区。分区包括序列化/反序列化、网络传输，以及磁盘上可能的缓冲。相比之下，不需要分区的操作可以快速扫描数据，并操作它看到的每一块数据。为部分重新计算检索相关的数据可以使用后一种方法完成。

检索相关数据的第一步是构建所需的相关数据的所有实体的集合。然后扫描主数据集，只发出集合中存在的实体的数据（每个任务都有该集合的副本）。在如 Hadoop 的批处理系统中，这将对应于一个 map-only 的作业。

因为内存有限，所以集合只能这么大。但一个称为布隆过滤器的数据结构可以使实体的集合更大。布隆过滤器是一个紧凑的数据结构，表示元素的集合并允许你询问它是否包含某个元素。布隆过滤器比集合更加紧凑，但作为权衡，布隆过滤器上的查询操作是概率性的。布隆过滤器有时会错误地告诉你一个元素存在于集合中，但它永远不会告诉你已经被添加到集合的一个元素不在集合中。因此，布隆过滤器有误报但没有漏报。

使用布隆过滤器来优化检索相关数据的过程如图 18-2 所示。如果使用布隆过滤器从主数据集中检索相关数据，那么会过滤掉绝大多数的主数据集。但是由于误报，将发出一些你不想检索到的数据，那么你可以在检索到的数据和所需的实体列表之间做一个连接 (Join)，来过滤掉误报的数据。连接需要分区，但是因为绝大多数的主数据集已经被过滤，所以去除误报的数据不是昂贵的操作。

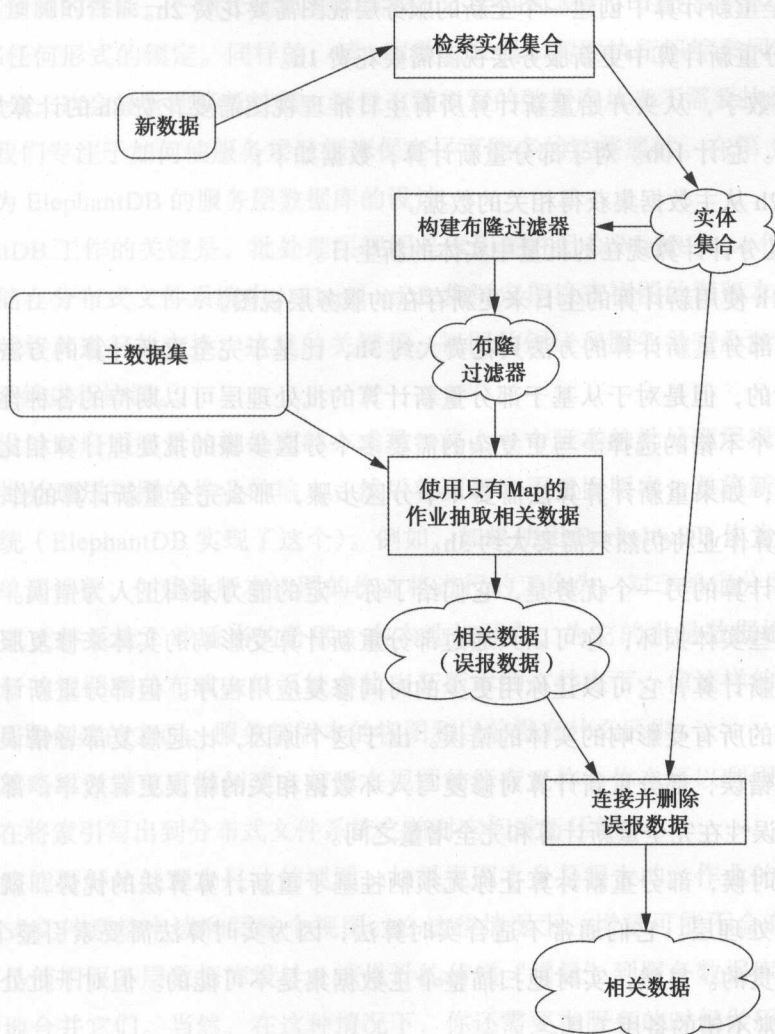


图 18-2 布隆连接

现在让我们做一些评估，看看与基于完全重新计算的批处理层相比，基于部分重新计算的批处理层改善了多少的延迟。假设计算生日推理需要一个带有分区的完整 MapReduce 作业，并且关于集群和数据存在以下事实：

- ❑ 主数据集包含 100TB 的数据。
- ❑ 基于部分重新计算方法的每个批量将有 50GB 的新数据。
- ❑ 带有分区的一个 MapReduce 作业在完整的主数据集上需要运行 8h。
- ❑ 一个 map-only 作业（不带有分区）在完整的主数据集上需要运行 2h（在 Map-Reduce 集群中，4 倍的速度差异是很典型的）。
- ❑ 在完全重新计算中创建一个全新的服务层视图需要花费 2h。
- ❑ 在部分重新计算中更新服务层视图需要花费 1h。

利用这些数字，从头开始重新计算所有生日推理视图需要花费 8h 的计算加上 2h 的构建服务层视图。总计 10h。对于部分重新计算，数据如下：

- ❑ 花费 2h 从主数据集获得相关的数据。
- ❑ 花费几分钟计算现在的批量中实体的新生日。
- ❑ 花费 1h 使用新计算的生日来更新存在的服务层视图。

所以基于部分重新计算的方法只花费大约 3h，比基于完全重新计算的方法快 70%。这些数字是估计的，但是对于从基于部分重新计算的批处理层可以期待的各种性能改进，它应该提供了一个不错的选择。与更复杂的需要多个分区步骤的批处理计算相比，这节省得要更多。例如，如果重新计算算法需要 4 个分区步骤，那么完全重新计算的作业需要 34h，而部分重新计算作业将仍然只需要大约 3h。

部分重新计算的另一个优势是，它们给了你一定的能力来纠正人为错误。如果写入了坏数据导致某些实体损坏，你可以只通过部分重新计算受影响的实体来修复服务层。比起完整的完全重新计算，它可以让你用更少的时间修复应用程序。但部分重新计算只帮助修复你可以识别的所有受影响的实体的错误。出于这个原因，比起修复部署错误代码导致视图损坏的相关错误，部分重新计算对修复写入坏数据相关的错误更有效率。部分重新计算的容忍人为错误性在完全重新计算和完全增量之间。

在适当的时候，部分重新计算让你无须牺牲基于重新计算算法的优势，就能拥有一个更少延迟的批处理层。它们通常不适合实时算法，因为实时算法需要索引整个主数据集，这将是非常昂贵的。显然，实时地扫描整个主数据集是不可能的。但对于批处理层，部分重新计算是非常不错的备用工具。

实现一个增量的批处理层

在增量的批处理层中，不管是做完全增量算法还是部分重新计算算法，增量的批处理层和基于重新计算的批处理层之间的主要区别是，需要更新服务层视图而不是从头开始创建它们。

构建类似于速度层的增量批处理层绝对是可行的，在适当的位置修改读 / 写数据库的视图。但由于不支持随机写，这将抵消服务层数据库的许多优点（第 10 章中讨论）：

- ❑ **鲁棒性**——没有随机写意味着代码库更简单，更不可能有错误。
 - ❑ **易于操作**——更少的移动部件意味着不必过多担心对数据库的操作——更少的配置和更少的出错。
 - ❑ **更可预测的性能**——由于随机写和读操作不会同时发生，因此没有必要担心数据库内部任何形式的锁定。同样的，读 / 写数据库偶尔需要执行压缩来回收索引未使用的部分，这会显著地降低性能，而没有随机写的数据库从来不需要执行这些操作。
- 所以让我们专注于如何使服务层数据库保存尽可能多的这些属性。在第 11 章中，你曾看到一个名为 ElephantDB 的服务层数据库的设计。

ElephantDB 工作的关键是，批处理层视图是索引的并且在 MapReduce 作业中被分区，这些索引存储在分布式文件系统中。ElephantDB 集群定期检查视图的新版本，并且一旦新版本可用，就将其进行热交换。这里的关键是，视图的创建和服务是完全独立的，并通过分布式文件系统进行协调。

扩展该设计以启用增量的批处理的方式是，将上一个版本的批处理层视图包含作为创建新版本的批处理层视图的作业的输入。然后将更新应用于旧版本，并将新版本写出到分布式文件系统（ElephantDB 实现了这个）。例如，如果使用 BerkeleyDB 作为索引系统，并在其中存储单词计数，创建新版本视图的作业将进行的工作为：对于给定分区视图的任务，它将从分布式文件系统下载适当的分区，在本地打开它，为它的批量数据增加单词计数，然后复制更新的视图到分布式文件系统中代表新版本的文件夹下。像这样的策略，所有增量都发生在视图创建的方面。服务新版本的视图和以前没有什么区别。

这样的策略可以避免重做创建之前版本视图的所有工作。你也可以利用批处理层的更高的延迟，在将索引写出到分布式文件系统之前对它们进行压缩。

这种策略能更好地处理小尺寸的视图。如果视图本身是很大的，作业的成本可能主要来自从分布式文件系统中读和写整个视图。在这些情况下，增量可能不会有太多的帮助。另一种策略是使用服务层数据库设计，该设计能传递“增量”到服务数据库，并且服务数据库能动态地合并它们。当然，在这种情况下，你还需要在服务的时候做压缩，因此服务层看起来将越来越像一个读 / 写数据库，并有许多与之相关的复杂性。

幸运的是，有一种方法可以最小化增量批处理层视图的大小，这样你就不会被迫为服务层使用增量策略或读 / 写数据库。相反，你可以保持服务层的优势，即视图的创建和服务是完全独立的。——这种策略就是拥有多个批处理层。

多批处理层

不是仅仅拥有一个批处理层和一个弥补批处理层延迟的速度层，而是可以有多个批处理层。例如，你可以有一个每月完成一次的基于完全重新计算的批处理层；然后你可以有一个增量批处理层——它只操作没有表示在完全重新计算的批处理层中的数据，可能每 6h 运行一次；然后你将有一个速度层，可以弥补所有没有表示在两个批处理层中的数据。

在基本 Lambda 架构中，批处理层放宽了对速度层的性能需求。同样的，对于多个批处理层，每一层都对其上面那一层放宽了需求。在提到的示例中，增量的批处理层只需要处理两个月的数据。这意味着它的视图可以保持得比总是表示所有数据的视图小得多。所以像这种基于旧的服务层视图得到全新的服务层视图的技术是可行的，因为复制视图的成本不会占主导地位。

拥有多个批处理层的另一个优势是，它可以有助于获得增量和重新计算两者的优势。增量的工作流可以更具性能但缺乏从错误中恢复的能力，而重新计算工作流可以赋予你这种能力。如果重新计算是系统中不断运行的一部分，那么你可以从任何错误中恢复过来。

系统每一层的延迟都直接影响了其上面一层的性能需求。所以对每一层的延迟是如何受到代码效率和分配给它们的资源量的影响有很好的理解，这是非常重要的。让我们看看这是如何表现出来的。

18.2.2 测量和优化批处理层的资源使用

事实证明，在运转批处理工作流的性能时，有很多违反直觉的动态变化。请思考如下示例，它们是基于现实世界情况的：

- ❑ 集群规模翻倍之后，批处理层的延迟从 30h 降低到 6h，有 80% 的改善。

- ❑ 不合适的重新配置使得 Hadoop 集群比以前多了 10% 的任务失败率。这导致批处理

工作流的运行时间从 8h 上升到 72h——性能降低为原来的 1/9。

你可能对此很难理解，但基本的动态变化是很容易阐释的。假设有一个需要运行 12h 的批处理工作流，那么每次迭代它就处理 12h 的数据。现在，假设你加强了工作流来做一些额外的分析，并且估计该分析将给当前的工作流增加 2h 的处理时间。现在你已经增加了工作流的运行时间——从操作 12h 的数据到 14h 的数据。这意味着下次工作流运行时，将处理 14h 的数据。因为下一次迭代有更多的数据，所以它将需要更长的时间来运行，这意味着下一次的迭代将有更多的数据，以此类推。

如果稳定的运行时间可以用一些非常简单的数学来确定，那么我们先写出批处理工作流的一次迭代的运行时间的方程。该方程将使用以下的变量：

- T ——工作流的运行时间，以小时为单位。
- O ——以小时为单位的工作流的开销。这是在工作流中独立于数据处理时间所花费的时间。这可以包括建立进程、在集群之间复制代码等。
- H ——迭代中处理数据的小时数。这里使用“小时”来测量数据量，因为它使得生成的等式非常简单。作为等式的一部分，它假定输入数据的速率是很稳定的。但是所得出的结论将不依赖于该变量。
- P ——动态处理时间。这是每小时的数据给工作流增加的处理时间的数值。如果每小时的数据增加了半小时的运行时间，那么 P 是 0.5。

基于这些定义，下面的方程是工作流一次迭代的运行时间的一种自然的表达

$$T = O + P \times H$$

当然， H 会随着工作流的每次迭代而发生变化，因为如果工作流比上一次迭代运行时间更短或更长，那么下一次迭代将分别有更少或更多的数据要处理。为了确定工作流的稳定运行时间，你需要确定工作流的运行时间等于其处理的数据的小时值的临界点。要做到这一点，你只需使 $T = H$ ，并解出 T

$$T = O + P \times H$$

$$T = \frac{O}{(1-P)}$$

如你所见，工作流的稳定运行时间与工作流的开销总量是成线性比例关系的。所以如果能减少 25% 的开销，那么工作流运行时间也会减少 25%。然而，工作流的稳定运行时间与动态处理时间 P 是非线性比例关系的。该方面的一个含义是，添加每台机器到集群中的性能收益是满足边际收益递减的。

如果 P 大于或等于 1 会发生什么？

你可能想知道如果动态处理时间 P ，大于或等于 1 将会发生什么。在这种情况下，工作流的每次迭代将比之前的迭代有更多的数据，因此批处理层将永远地被甩在后面。所以保持 P 小于 1 是非常重要的。

使用该方程，前面描述的违反直觉的情况将更有意义。下面来看当集群规模加倍时，稳定运行时间将会发生什么变化。当这种情况发生时，动态处理时间 P 大约被减半，因为你现在可以并行两倍进行处理（从技术上讲，协调所有这些机器的开销也稍有增加，但让我们忽略该项）。如果 T_1 是集群规模加倍之前的稳定运行时间， T_2 是之后的稳定运行时间，那么你将得到以下两个方程

$$T1 = \frac{O}{(1-P)}$$

$$T2 = \frac{O}{(1-P/2)}$$

$T2/T1$ 的结果为

$$\frac{T2}{T1} = \frac{1-P}{(2-P)}$$

该方程的曲线图如图 18-3 所示。

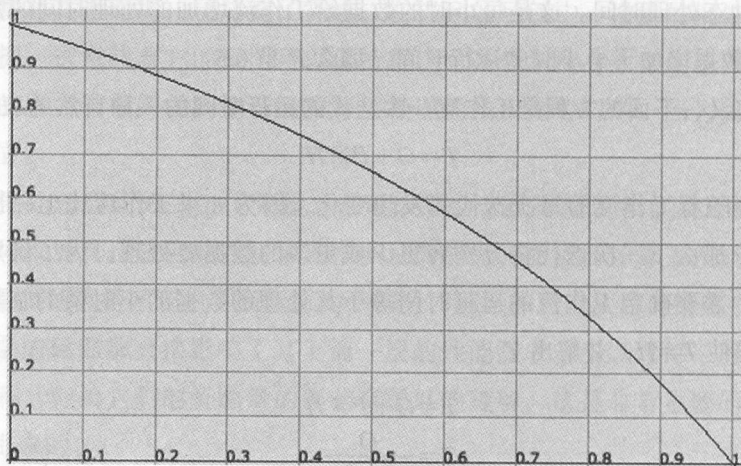


图 18-3 加倍的集群规模的性能影响

这张图足以说明一切。如果 P 值很小，比如每小时的数据花费 6min 的处理时间，那么加倍的集群规模几乎不会影响到运行时间。这是有道理的，因为运行时间由开销占主要地位，并不受加倍的集群规模影响。

但是如果 P 值很大，比如每小时的数据要花费 54min 的动态时间，那么加倍的集群规模将导致新的运行时间是原始运行时间的 18%，即加速 82%！在这种情况下，下一次迭代将完成得更快，从而导致下一次迭代的数据更少，那么它将完成得更快。这种积极的循环最终稳定在 82% 的速度提升。

现在来考虑失败率的增加对稳定运行时间的影响。10% 的任务失败率意味着你将需要执行额外 11% 的任务来处理数据。（如果有 100 个任务，其中 10 个任务失败，你会重试这 10 个任务。然而平均 10 个任务中的 1 个也将会失败，所以你也需要重试这个任务。）因为任务依赖于所拥有的数据量，所以这意味着处理一个小时数据的时间 (P) 将增加 11%。

在最后的分析中，我们称 $T1$ 是失败发生之前的运行时间， $T2$ 是失败发生之后的运行时间

$$T1 = \frac{O}{(1-P)}$$

$$T2 = \frac{O}{(1-1.11 \times P)}$$

$T2/T1$ 的结果为

$$\frac{T2}{T1} = \frac{(1-P)}{(1-1.11 \times P)}$$

该方程的曲线图如图 18-4 所示。

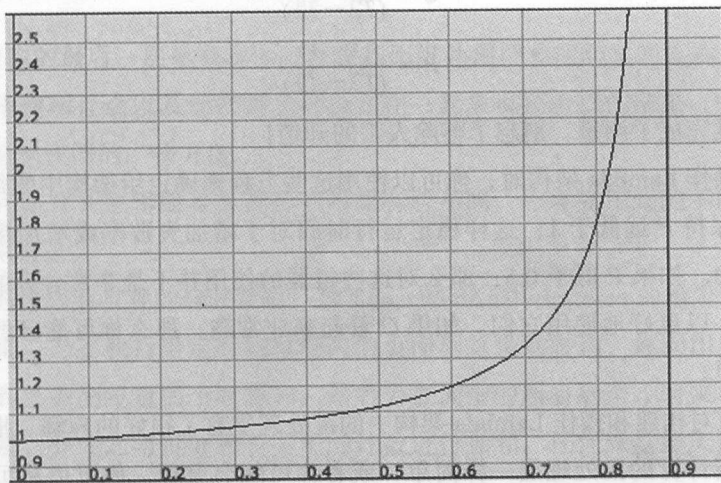


图 18-4 增加 10% 失败率的性能影响

如图 18-4 所示, P 越接近 1, 失败率增加时, 稳定运行时间呈戏剧性变化。这就是所谓的“10% 失败率的增加会导致性能降低至原来的 1/9”。保持 P 小于 1 是很重要的, 这样面对集群将经历的自然变化, 运行时间将是稳定的。根据该图 18-4 可知, P 小于 0.7 似乎是很安全的。

通过优化代码, 你可以控制 O 和 P 的值。此外, 你可以通过分配给批处理工作流的资源 (如物理机) 来控制 P 的值。 P 的临界值是 0.5。当 P 高于 0.5 时, 添加 1% 的机器将减少超过 1% 的延迟, 这是比较合算的。当 P 低于 0.5 时, 添加 1% 的机器将减少不到 1% 的延迟, 这在提高成本效益方面是有问题的。

为了给工作流测量 O 和 P 的值, 你可能会试图在零数据上运行工作流。这将使方程变为 $T=O+P \times 0$, 让你很容易地解出 O 。然后, 你可以使用该值来解出方程 $T=O/(1-P)$ 中的 P 。但是这种方法往往是不准确的。例如在 Hadoop 上, 一个作业通常有比集群任务槽更多的任务。一个作业将花费几分钟来开始运行并达到全速度来利用集群上所有可用的任务槽。所花费的开始运行的时间通常是一个固定的时间, 所以它可以被 O 变量所捕获。当

使用少量的数据运行作业时，作业将在利用整个集群之前结束，这会使得对 O 的测量出现偏差。

测量 O 和 P 更好的一种方法是人为地在工作流中引入开销，比如在代码中添加一个 `sleep(1 hour)` 的调用。工作流的运行时间一旦稳定，你将有两个测量 $T1$ 和 $T2$ ，分别对应增加开销之前和之后。最终得到以下 O 和 P 值的方程

$$O = \frac{T1}{(T2 - T1)}$$

$$P = \frac{(1 - I)}{(T2 - T1)}$$

当然，一旦完成了测量，别忘了移除人工的开销！

在构建和操作 Lambda 架构时，你可以使用这些方程来确定给架构中的每个批处理层多少资源。你想保持 P 远低于 1，这样稳定运行时间对于增加失败率或增加接收到的数据的速率都是弹性的。如果 P 低于 0.5，那么对这些机器的使用并不是非常合算的，所以你应该考虑分配它们，以更好地使用它们。如果 O 看起来非常高，那么你可能已经确定了工作流中的另一个瓶颈。

现在你应该对构建和操作 Lambda 架构中的批处理层有了很好的理解。批处理层的设计可以与基于重新计算的批处理层一样简单，或者你可能会发现，你可以从可能结合了基于重新计算批处理层的增量批处理层中受益。下面继续讨论 Lambda 架构的速度层。

18.3 速度层

因为服务层以高延迟进行更新，所以它总是过时几个小时的。但服务层中的视图代表了绝大多数已有的数据——没有被表示的数据是上次服务层更新后到达的数据。剩下的就是让你能实时查询，来补偿那些最后几个小时的数据。这就是速度层的目的。

在速度层做取舍时，你倾向于性能——使用增量算法而不是重新计算算法，使用可变的读/写数据库而不是服务层中首选的数据库类型。之所以这样做，是出于低延迟的需求，而且这些缺乏容忍人为错误的方法最终并不重要。因为服务层会不断覆盖速度层，所以速度层中的错误可以很容易地被纠正。

在传统架构中，通常只有一个与速度层大致相当的层。但由于没有支撑它的批处理层，因此它很容易遇到会导致数据损坏的错误。此外，操作 PB 级别的读/写数据库所面临的业务挑战是巨大的。Lambda 架构中的速度层对于这些业务挑战大多是不予约束的，因为批处理层和服务层极大程度地放宽了它们的需求。因为速度层只表示最近的数据，其视图可以

保持得非常小，避免了上述的业务挑战。

在第 12~17 章中，你知道了构建速度层的错综复杂和各种变化，包括队列化、同步和异步速度层的对比以及一次一个流处理和微批量流处理的对比。你知道了速度层使用近似方法来减少复杂性，提高性能，或同时解决这两个问题。

18.4 查询层

Lambda 架构的最后一层是查询层，负责利用批处理层和实时视图来响应查询。它确定了从每个视图中使用什么以及如何将它们合并在一起得到合适的结果。每个查询表述为批处理视图和实时视图的一些方法。

查询中使用的合并逻辑随着查询的不同而有所不同。你可能用到的不同技术能通过几个示例很好地阐述。

面向时间的查询有直观的合并策略，如 SuperWebAnalytics.com 的给定时间范围内的页面浏览量的查询。为了执行给定时间范围内的页面浏览量的查询，你利用批处理层已经完成的数据，得到页面浏览量的小时总和；然后为查询中的所有剩余小时来检索速度层视图中的页面浏览量，并使之与批处理视图中的计数进行加和。任何像这样天然以时间分割的查询将有类似的合并策略。

你可为本章前面引入的生日推理问题采取一种不同的方法。该方法如下：

- ❑ 批处理层运行一个算法，该算法将妥善处理混乱的数据并选择一个日期的范围作为输出。和范围一起发出的还有进入计算该范围的年龄样本的数量。
- ❑ 速度层通过使用每个年龄样本缩小范围，来增量地计算出一个范围。如果一个年龄样本将消除作为生日的任何可能日期，那么该样本被忽略。这个增量策略是快速且简单的，但它没有处理混乱的数据。不过没关系，因为这是由批处理层处理的。速度层还存储进入计算其范围的样本的数量。
- ❑ 为了响应查询，使用批处理和速度层相关的样本数量来检索它们的范围。如果两个范围合并在一起没有消除所有可能的日期，那么它们被合并到尽可能最小的范围内；否则，使用更高样品数量的范围作为结果。

该生日推理的策略使视图很简单并能处理所有适当的情况。对于该系统是新加入的人来说，将使用速度层中的增量算法进行合适的处理。它并没有像批处理层那样处理混乱的数据，但它足够好到批处理层以后可以做更复杂的分析。该策略还能处理新数据的突然爆发。如果你突然地添加了一群年龄样本到该系统中，那么将使用速度层的结果而不是批处

理层结果，因为它是基于更多数据的。当然，批处理层总是重新计算生日范围，所以随着时间的推移结果将越来越准确。可能你选择为生日推理使用的实现会有所变化，但你应该理解其思想。

从这些例子中你可以很明显地知道，为了便于合并，视图必须是结构化的。这对于像给定时间范围内的页面浏览量之类面向时间的查询是很自然的，但生日推理的示例特别添加了样本数量到视图中，以帮助合并。如何构建视图使得它们可以合并，是实现 Lambda 架构时必须做出的设计选项之一。

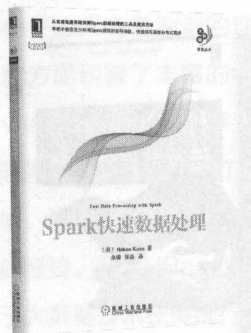
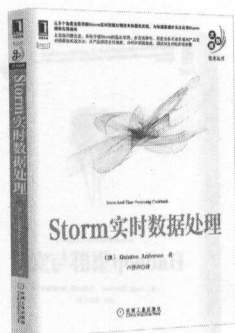
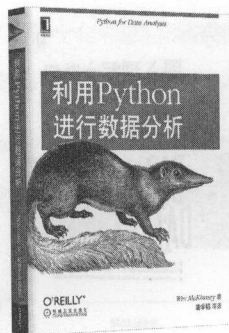
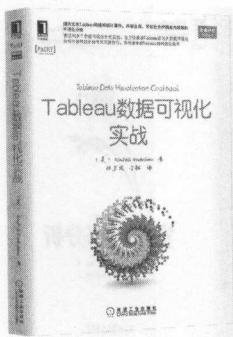
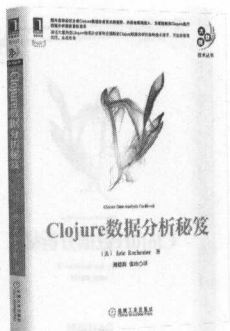
18.5 总结

Lambda 架构是从基本原理——数据问题的一般表述方式作为你已知的所有数据的函数——出发的结果，并实现了一些必需的需求，如容忍人为错误、水平可扩展性、低延迟读取和低延迟更新。我们已经探讨了 Lambda 架构，并使用了很多工具来提供核心原则的实例，比如 Hadoop、JCascalog、Kafka、Cassandra 和 Storm。我们希望大家清楚这些工具不是 Lambda 架构的必要部分。我们完全相信这些工具随着时间的推移将发生变化和发展，但 Lambda 架构的原则总是不变的。

在许多方面，Lambda 架构都超越了当前可用的工具。尽管现如今实现 Lambda 架构是非常可行的——通过本书探讨实现不同层的细节，我们尝试来说明这件事情——它当然可以更容易地实现。只有少数几个专门设计用于服务层的数据库，拥有速度层数据库将是很棒的，它可以更容易地处理视图中不再需要的过期的部分。幸运的是，构建这些工具比构建广泛的传统读/写数据库更容易，因此我们期望这些漏洞随着越来越多的人采用 Lambda 架构而得以填补。与此同时，你会发现在 Lambda 架构中，你会为不同的角色改变传统数据库的用途，并自己做一些研发使这些数据库更合用。

当第一次遇到大数据问题和大数据生态系统的工具时，人们很容易混淆和不知所措。对熟悉的关系型数据库的向往是可以理解的，关系型数据库在过去几十年里作为一个产业已经变得如此让人熟悉。希望通过学习 Lambda 架构，你已经知道了构建大数据系统可以比构建基于传统架构的系统简单得多。Lambda 架构完全解决了规范化和反规范化问题以及一些困扰传统架构的问题，并且它也有固有的容忍人为错误属性——我们认为这是不可妥协的属性。此外，它避免了基于整体读/写数据库的架构所带来的过多的复杂性。因为它基于所有数据上的方法，所以 Lambda 架构就其本性而言是通用的，这会给你带来积极处理任何数据问题的信心。

推荐阅读



■ Clojure数据分析秘笈

作者: Eric Rochester

ISBN: 978-7-111-47326-8

定价: 59.00元

■ 利用Python进行数据分析

作者: Wes McKinney

ISBN: 978-7-111-43673-7

定价: 89.00元

■ Splunk大数据分析

作者: Peter Zadrozny 等

ISBN: 978-7-111-46429-7

定价: 69.00元

■ Tableau数据可视化实战

作者: Ashutosh Nandeshwar

ISBN: 978-7-111-47283-4

定价: 39.00元

■ Storm实时数据处理

作者: Quinton Anderson

ISBN: 978-7-111-46663-5

定价: 49.00元

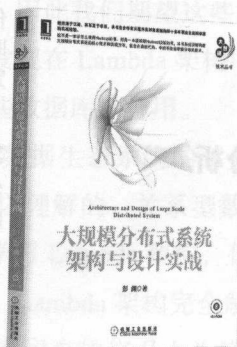
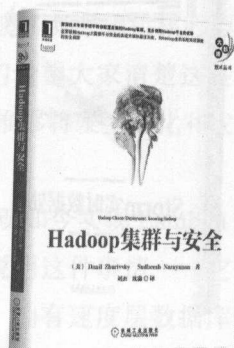
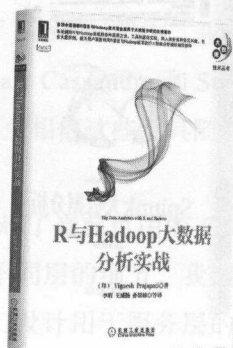
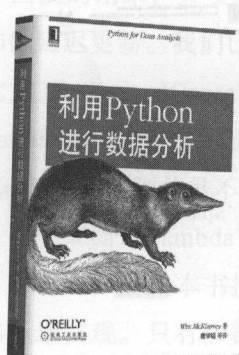
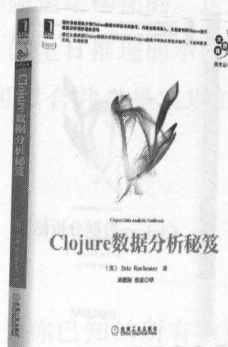
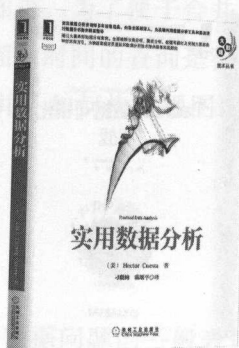
■ Spark快速数据处理

作者: Holden Karau

ISBN: 978-7-111-46311-5

定价: 29.00元

推荐阅读



作者简介

Nathan Marz Cascalog和Storm的创始人。在2011年Twitter收购社交媒体数据分析公司BackType前，他是BackType首席工程师。在Twitter，他建立了流计算团队，提供和开发共享基础设施，为整个公司的关键实时应用提供支持。他目前是Stealth startup的创始人。

James Warren Storm8的分析架构师，精通大数据处理、机器学习和科学计算。

译者简介

马延辉 资深Hadoop技术专家，对Hadoop生态系统相关技术有着深刻的理解，在Hadoop开发和运维方面积累了丰富的经验。曾就职于阿里、Answers.com、暴风等互联网公司，从事Hadoop相关的研发和运维工作，对大数据技术的企业级落地、研发、运维和管理有着深刻的理解和丰富的实战经验。开源HBase监控工具Ella作者。现在致力于大数据技术在传统行业的落地和大数据技术的普及和推广。

向磊 前暴风影音数据平台架构师，目前在Admaster担任基础架构部架构师，惠普中国Hadoop相关课程讲师。开源项目EasyHadoop、phpHiveAdmin作者，对Hadoop及其周边生态系统的底层运维及开发、集群自动化运维、网络架构设计、集群安全、性能优化、嵌入式编程有较为深入的了解。

魏东琦 博士，长期从事软件研发工作，现就职于中国地质调查局西安地质调查中心，参加、承担过多项科研项目。现致力于地质行业与大数据技术融合的相关研究工作。

超越了个别工具或平台。任何从事大数据系统工作的人都需要阅读。

—— **Jonathan Esterhazy**, Groupon

一次全面的、样例驱动的Lambda架构之旅，由Lambda架构的发起人为您指导。

—— **Mark Fisher**, Pivotal

内含只有在经历许多大数据项目后才能获得的智慧。这是一本必须阅读的书。

—— **Pere Ferrera Bertran**, Datasalt

在批处理和近似实时处理中，简化数据管道的实际指南。

—— **Alex Holmes**, 《Hadoop实践》作者

近年来，互联网技术发展迅猛，从电子交易记录、社交网络数据分析到地震分析、分子建模，各行各业应用大数据系统的范围日益拓宽，所涉及的数据量日益“臃肿”，对处理速度的要求也日益提高，这就需要用基于硬件集群构建的架构进行存储和处理。但这种架构在提供便利的同时，也引入了大多数开发者并不熟悉的、困扰传统架构的复杂性问题。

本书教你使用一种专门设计用来获取和分析网络规模数据的架构去构建大数据系统——Lambda架构，它是一种可扩展的、易于理解的、可以被小团队用来构建和运行大数据系统的方法。除了与你分享Lambda架构的相关知识，本书还给出了相应的示例，将“理论应用于实践”，助你更好地“认识”Lambda架构，更好地将其应用到工作中。



MANNING

投稿热线: (010) 88379604

客服热线: (010) 88379426 88361066

购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com

网上购书: www.china-pub.com

数字阅读: www.hzmedia.com.cn



上架指导: 计算机/大数据

ISBN 978-7-111-55294-9



9 787111 552949 >

定价: 79.00元